OCI

Mimanshu Maheshwari

March 27, 2025

Contents

List of Figures

Listings

Chapter 1

Introduction

1.1 What is OCI?

Oracle Cloud Infrastructure (OCI) is a cloud computing service offered by Oracle that provides Infrastructure as a Service (IaaS), Platform as a Service (PaaS), Software as a Service (SaaS), and Data as a Service (DaaS) solutions. It is designed to support high-performance computing, enterprise workloads, and cloud-native applications. OCI provides services such as compute, storage, networking, databases, security, and Artificial Intelligence (AI)/Machine Learning (ML) tools.

Chapter 2

Terraform

2.1 What is IaC?

Infrastructure as Code (IaC) tools allow you to manage infrastructure with configuration files rather than through a graphical user interface. IaC allows you to build, change, and manage your infrastructure in a safe, consistent, and repeatable way by defining resource configurations that you can version, reuse, and share. Terraform is HashiCorp's infrastructure as code tool. It lets you define resources and infrastructure in human-readable, declarative configuration files, and manages your infrastructure's lifecycle. Using Terraform has several advantages over manually managing your infrastructure:

- Terraform can manage infrastructure on multiple cloud platforms.
- The human-readable configuration language helps you write infrastructure code quickly.
- Terraform's state allows you to track resource changes throughout your deployments.
- You can commit your configurations to version control to safely collaborate on infrastructure.

2.2 Manage any infrastructure

Terraform plugins called providers let Terraform interact with cloud platforms and other services via their application programming interfaces (APIs). HashiCorp and the Terraform community have written over 1,000 providers to manage resources on Amazon Web Services (AWS), Azure, Google Cloud Platform (GCP), Kubernetes, Helm, GitHub, Splunk, and DataDog, just to name a few. Find providers for many of the platforms and services you already use in the Terraform Registry. If you don't find the provider you're looking for, you can write your own.

2.3 Standardize your deployment workflow

Providers define individual units of infrastructure, for example compute instances or private networks, as resources. You can compose resources from different providers into reusable Terraform configurations called modules, and manage them with a consistent language and workflow. Terraform's configuration language is declarative, meaning that it describes the desired end-state for your infrastructure, in contrast to procedural programming languages that require step-by-step instructions to perform tasks. Terraform providers automatically calculate dependencies between resources to create or destroy them in the correct order.

To deploy infrastructure with Terraform(2.1):

- **Scope**: Identify the infrastructure for your project.
- **Author**: Write the configuration for your infrastructure.
- **Initialize**: Install the plugins Terraform needs to manage the infrastructure.

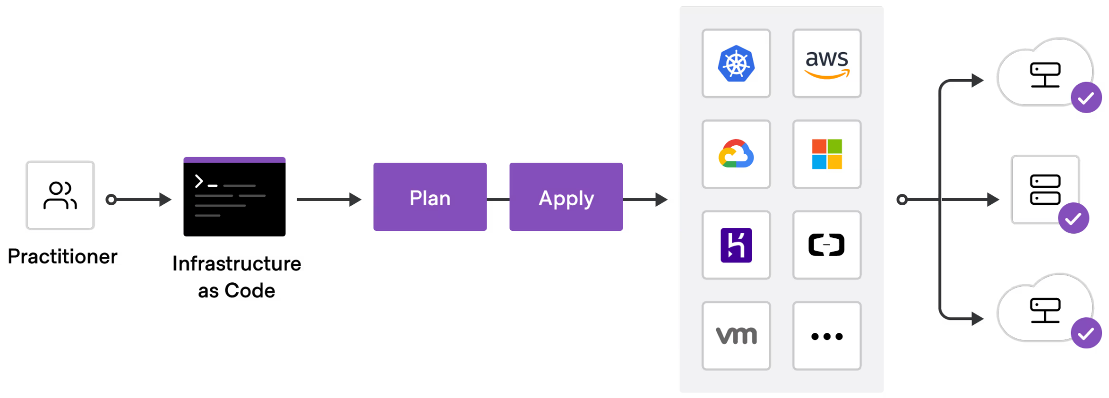

Figure 2.1: Terraform process flow

- **Plan**: Preview the changes Terraform will make to match your configuration.
- **Apply**: Make the planned changes.

2.4 Track your infrastructure

Terraform keeps track of your real infrastructure in a state file, which acts as a source of truth for your environment. Terraform uses the state file to determine the changes to make to your infrastructure so that it will match your configuration.

```
terraform {
  required_providers {
    docker ={
      source ="kreuzwerker/docker"
      version ="~> 3.0.2"
    }
  }
  required_version ="~> 1.7"
}
```

Listing 2.1: terraform.tf

This file includes the terraform block, which defines the provider and Terraform versions you will use with this project.

```
provider "docker" {}

resource "docker_image" "nginx" {
  name ="nginx:latest"
  keep_locally =false
}

resource "docker_container" "nginx" {
  image =docker_image.nginx.image_id
  name ="tutorial"
  ports {
    internal =80
    external =8000
  }
}
```

Listing 2.2: main.tf

```
terraform init
```

Provision the NGINX server container with apply. When Terraform asks you to confirm, type yes and press ENTER.

```
terraform apply
```

Run `docker ps` to view the NGINX container running in Docker via Terraform.

```
docker ps
```

```
terraform destroy
```

To stop the container and destroy the resources created in this tutorial, run `terraform destroy`. When Terraform asks you to confirm, type yes and press ENTER.

2.5 CLI Usage

Create a directory named `learn-terraform-docker-container`.

```
mkdir learn-terraform-docker-container
```

This working directory houses the configuration files that you write to describe the infrastructure you want Terraform to create and manage. When you initialize and apply the configuration here, Terraform uses this directory to store required plugins, modules (pre-written configurations), and information about the real infrastructure it created. Navigate into the working directory.

```
cd learn-terraform-docker-container
```

In the working directory, create a file called main.tf and paste the following Terraform configuration into it.

```

terraform {
  required_providers {
    docker ={
      source ="kreuzwerker/docker"
      version ="~> 3.0.1"
    }
  }
}

provider "docker" {
  # remove host for mac or linux
  host ="npipe:////.//pipe//docker_engine"
}

resource "docker_image" "nginx" {
  name ="nginx"
  keep_locally =false
}

resource "docker_container" "nginx" {
  image =docker_image.nginx.image_id
  name ="tutorial"

  ports {
    internal =80
    external =8000
  }
}
```

Listing 2.3: main.tf

Initialize the project, which downloads a plugin called a provider that lets Terraform interact with Docker.

```
terraform init
```

Provision the NGINX server container with apply. When Terraform asks you to confirm type yes and press ENTER.

```
terraform apply
```

Verify the existence of the NGINX container by visiting localhost:8000 in your web browser or running `docker ps` to see the container. To stop the container, run terraform destroy.

```
terraform destroy
```

You've now provisioned and destroyed an NGINX webserver with Terraform.