Chapter 7. Working with Multiple Providers

So far, almost every single example in this book has included just a single `provider` block:

```
provider "aws" {
  region = "us-east-2"
}
```

This `provider` block configures your code to deploy to a single AWS region in a single AWS account. This raises a few questions:

- What if you need to deploy to multiple AWS regions?
- What if you need to deploy to multiple AWS accounts?
- What if you need to deploy to other clouds, such as Azure or GCP?

To answer these questions, this chapter takes a deeper look at Terraform providers:

- Working with one provider
- Working with multiple copies of the same provider
- Working with multiple different providers

Working with One Provider

So far, you've been using providers somewhat "magically." That works well enough for simple examples with one basic provider, but if you want to work with multiple regions, accounts, clouds, etc., you'll need to go deeper. Let's start by taking a closer look at a single provider to better understand how it works:

- What is a provider?
- How do you install providers?
- How do you use providers?

What Is a Provider?

When I first introduced providers in Chapter 2, I described them as the *platforms* Terraform works with: e.g., AWS, Azure, Google Cloud, DigitalOcean, etc. So how does Terraform interact with these platforms?

Under the hood, Terraform consists of two parts:

Core

This is the `terraform` binary, and it provides all the basic functionality in Terraform that is used by all platforms, such as a command-line interface (i.e., `plan`, `apply`, etc.), a parser and interpreter for Terraform code (HCL), the ability to build a dependency graph from resources and data sources, logic to read and write state files, and so on. Under the hood, the code is written in Go and lives in an open source GitHub repo owned and maintained by HashiCorp.

Providers

Terraform providers are *plugins* for the Terraform core. Each plugin is written in Go to implement a specific interface, and the Terraform core knows how to install and execute the plugin. Each of these plugins is designed to work with some platform in the outside world, such as AWS, Azure, or Google Cloud. The Terraform core communicates with plugins via *remote procedure calls* (RPCs), and those plugins, in turn, communicate with their corresponding platforms via the network (e.g., via HTTP calls), as shown in Figure 7-1.

The code for each plugin typically lives in its own repo. For example, all the AWS functionality you've been using in the book so far comes from a plugin called the Terraform AWS Provider (or just AWS Provider for short) that lives in its own repo. Although HashiCorp created most of the initial providers, and still helps to maintain many of them, these days, much of the work for each provider is done by the company that owns the underlying platform: e.g., AWS employees work on the AWS Provider, Microsoft employees work on the Azure provider, Google employees work on the Google Cloud provider, and so on.

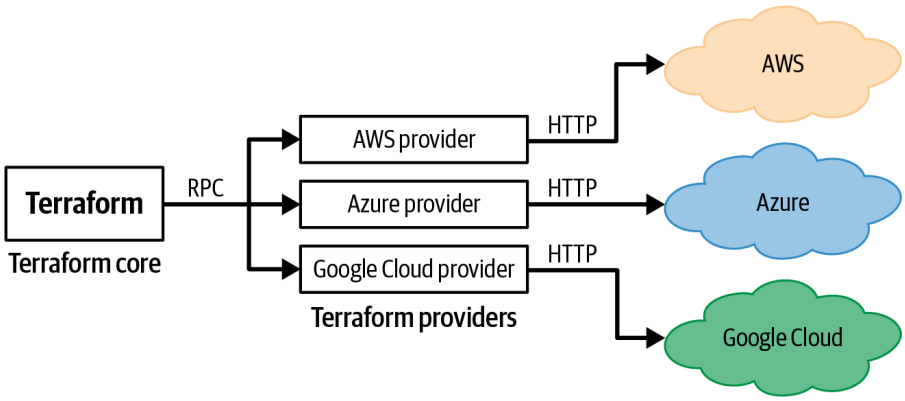

Figure 7-1. The interaction between the Terraform core, providers, and the outside world.

Each provider claims a specific prefix and exposes one or more resources and data sources whose names include that prefix: e.g., all the resources and data sources from the AWS Provider use the `aws_` prefix (e.g., `aws_instance`, `aws_autoscaling_group`, `aws_ami`), all the resources and data sources from the Azure provider use the `azurerm_` prefix (e.g., `azurerm_virtual_machine`, `azurerm_virtual_machine_scale_set`, `azurerm_image`), and so on.

How Do You Install Providers?

For official Terraform providers, such as the ones for AWS, Azure, and Google Cloud, it's enough to just add a `provider` block to your code:[1]

```
provider "aws" {
  region = "us-east-2"
}
```

As soon as you run `terraform init`, Terraform automatically downloads the code for the provider:

```
$ terraform init

Initializing provider plugins...
- Finding hashicorp/aws versions matching "4.19.0"...
- Installing hashicorp/aws v4.19.0...
- Installed hashicorp/aws v4.19.0 (signed by HashiCorp)
```

This is a bit magical, isn't it? How does Terraform know what provider you want? Or which version you want? Or where to download it from? Although it's OK to rely on this sort of magic for learning and experimenting, when writing production code, you'll probably want a bit more control over how Terraform installs providers. Do this by adding a `required_providers` block, which has the following syntax:

```
terraform {
  required_providers {
    <LOCAL_NAME> = {
      source  = "<URL>"
      version = "<VERSION>"
    }
  }
}
```

where:

LOCAL_NAME

: This is the *local name* to use for the provider in this module. You must give each provider a unique name, and you use that name in the `provider` block configuration. In almost all cases, you'll use the *preferred local name* of that provider: e.g., for the AWS Provider, the preferred local name is `aws`, which is why you write the provider block as `provider "aws" { … }`. However, in rare cases, you may end up with two providers that have the same preferred local name—e.g., two providers that both deal with HTTP requests and have a preferred local name of `http`—so you can use this local name to disambiguate between them.

URL

: This is the URL from where Terraform should download the provider, in the format `[<HOSTNAME>/]<NAMESPACE>/<TYPE>`, where `HOSTNAME` is the hostname of a Terraform Registry that distributes the provider, `NAMESPACE` is the organizational namespace (typically, a company name), and `TYPE` is the name of the platform this provider manages (typically, `TYPE` is the preferred local name). For example, the full URL for the AWS Provider, which is hosted in the public Terraform Registry, is `registry.terraform.io/hashicorp/aws`. However, note that `HOSTNAME` is optional, and if you omit it, Terraform will by default download the provider from the public Terraform Registry, so the shorter and more common way to specify the exact same AWS Provider URL is `hashicorp/aws`. You typically only include `HOSTNAME` for custom providers that you're downloading from private Terraform Registries (e.g., a private Registry you're running in Terraform Cloud or Terraform Enterprise).

VERSION

: This is a version constraint. For example, you could set it to a specific version, such as `4.19.0`, or to a version range, such as `> 4.0, < 4.3`. You'll learn more about how to handle versioning in Chapter 8.

For example, to install version 4.x of the AWS Provider, you can use the following code:

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.0"
    }
```

```
  }
}
```

So now you can finally understand the magical provider installation behavior you saw earlier. If you add a new `provider` block named `foo` to your code, and you don't specify a `required_providers` block, when you run `terraform init`, Terraform will automatically do the following:

- Try to download provider `foo` with the assumption that the `HOSTNAME` is the public Terraform Registry and that the `NAMESPACE` is `hashicorp`, so the download URL is `registry.terraform.io/hashicorp/foo`.
- If that's a valid URL, install the latest version of the `foo` provider available at that URL.

If you want to install any provider not in the `hashicorp` namespace (e.g., if you want to use providers from Datadog, Cloudflare, or Confluent, or a custom provider you built yourself), or you want to control the version of the provider you use, you will need to include a `required_providers` block.

ALWAYS INCLUDE REQUIRED_PROVIDERS

As you'll learn in Chapter 8, it's important to control the version of the provider you use, so I recommend *always* including a `required_providers` block in your code.

How Do You Use Providers?

With this new knowledge about providers, let's revisit how to use them. The first step is to add a `required_providers` block to your code to specify which provider you want to use:

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.0"
    }
  }
}
```

Next, you add a `provider` block to configure that provider:

```
provider "aws" {
  region = "us-east-2"
}
```

So far, you've only been configuring the `region` to use in the AWS Provider, but there are many other settings you can configure. Always check your provider's documentation for the details: typically, this documentation lives in the same Registry you use to download the provider (the one in the `source` URL). For example, the documentation for the AWS Provider is in the public Terraform Registry. This documentation will typically explain how to configure the provider to work with different users, roles, regions, accounts, and so on.

Once you've configured a provider, all the resources and data sources from that provider (all the ones with the same prefix) that you put into your code will automatically use that configuration. So, for example, when you set the region in the `aws` provider to `us-east-2`, all the `aws_` resources to your code will automatically deploy into `us-east-2`.

But what if you want some of those resources to deploy into `us-east-2` and some into a different region, such as `us-west-1`? Or what if you want to deploy some resources to a completely different AWS account? To do that, you'll have to learn how to configure multiple copies of the same provider, as discussed in the next section.

Working with Multiple Copies of the Same Provider

To understand how to work with multiple copies of the same provider, let's look at a few of the common cases where this comes up:

- Working with multiple AWS regions
- Working with multiple AWS accounts
- Creating modules that can work with multiple providers

Working with Multiple AWS Regions

Most cloud providers allow you to deploy into datacenters ("regions") all over the world, but when you configure a Terraform provider, you typically configure it to deploy into just one of those regions. For example, so far you've been deploying into just a single AWS region, `us-east-2`:

```
provider "aws" {
  region = "us-east-2"
```

```
}
```

What if you wanted to deploy into multiple regions? For example, how could you deploy some resources into `us-east-2` and other resources into `us-west-1`? You might be tempted to solve this by defining two `provider` configurations, one for each region:

```
provider "aws" {
  region = "us-east-2"
}

provider "aws" {
  region = "us-west-1"
}
```

But now there's a new problem: How do you specify which of these `provider` configurations each of your resources, data sources, and modules should use? Let's look at data sources first. Imagine you had two copies of the `aws_region` data source, which returns the current AWS region:

```
data "aws_region" "region_1" {
}

data "aws_region" "region_2" {
}
```

How do you get the `region_1` data source to use the `us-east-2` provider and the `region_2` data source to use the `us-west-1` provider? The solution is to add an alias to each provider:

```
provider "aws" {
  region = "us-east-2"
  alias  = "region_1"
}

provider "aws" {
  region = "us-west-1"
  alias  = "region_2"
}
```

An *alias* is a custom name for the `provider`, which you can explicitly pass to individual resources, data sources, and modules to get them to use the configuration in that particular provider. To tell those `aws_region` data sources to use a specific provider, you set the `provider` parameter as follows:

```
data "aws_region" "region_1" {
  provider = aws.region_1
}

data "aws_region" "region_2" {
  provider = aws.region_2
}
```

Add some output variables so you can check that this is working:

```
output "region_1" {
  value       = data.aws_region.region_1.name
  description = "The name of the first region"
}

output "region_2" {
  value       = data.aws_region.region_2.name
  description = "The name of the second region"
}
```

And run `apply`:

```
$ terraform apply

(...)

Outputs:

region_1 = "us-east-2"
region_2 = "us-west-1"
```

And there you go: each of the `aws_region` data sources is now using a different provider and, therefore, running against a different AWS region. The same technique of setting the `provider` parameter works with resources too. For example, here's how you can deploy two EC2 Instances in different regions:

```
resource "aws_instance" "region_1" {
  provider = aws.region_1

  # Note different AMI IDs!!
  ami           = "ami-0fb653ca2d3203ac1"
  instance_type = "t2.micro"
}

resource "aws_instance" "region_2" {
  provider = aws.region_2
```

```
  # Note different AMI IDs!!
  ami           = "ami-01f87c43e618bf8f0"
  instance_type = "t2.micro"
}
```

Notice how each `aws_instance` resource sets the `provider` parameter to ensure it deploys into the proper region. Also, note that the `ami` parameter has to be different on the two `aws_instance` resources: that's because AMI IDs are unique to each AWS region, so the ID for Ubuntu 20.04 in `us-east-2` is different than for Ubuntu 20.04 in `us-west-1`. Having to look up and manage these AMI IDs manually is tedious and error prone. Fortunately, there's a better alternative: use the `aws_ami` data source that, given a set of filters, can find AMI IDs for you automatically. Here's how you can use this data source twice, once in each region, to look up Ubuntu 20.04 AMI IDs:

```
data "aws_ami" "ubuntu_region_1" {
  provider = aws.region_1

  most_recent = true
  owners      = ["099720109477"] # Canonical

  filter {
    name   = "name"
    values = ["ubuntu/images/hvm-ssd/ubuntu-focal-20.04-amd64-server-*"]
  }
}

data "aws_ami" "ubuntu_region_2" {
  provider = aws.region_2

  most_recent = true
  owners      = ["099720109477"] # Canonical

  filter {
    name   = "name"
    values = ["ubuntu/images/hvm-ssd/ubuntu-focal-20.04-amd64-server-*"]
  }
}
```

Notice how each data source sets the `provider` parameter to ensure it's looking up the AMI ID in the proper region. Go back to the `aws_instance` code and update the `ami` parameter to use the output of these data sources instead of the hardcoded values:

```
resource "aws_instance" "region_1" {
  provider = aws.region_1

  ami           = data.aws_ami.ubuntu_region_1.id
```

```
  instance_type = "t2.micro"
}

resource "aws_instance" "region_2" {
  provider = aws.region_2

  ami           = data.aws_ami.ubuntu_region_2.id
  instance_type = "t2.micro"
}
```

Much better. Now, no matter what region you deploy into, you'll automatically get the proper AMI ID for Ubuntu. To check that these EC2 Instances are really deploying into different regions, add output variables that show you which availability zone (each of which is in one region) each instance was actually deployed into:

```
output "instance_region_1_az" {
  value       = aws_instance.region_1.availability_zone
  description = "The AZ where the instance in the first region deployed"
}

output "instance_region_2_az" {
  value       = aws_instance.region_2.availability_zone
  description = "The AZ where the instance in the second region deployed"
}
```

And now run `apply`:

```
$ terraform apply

(...)

Outputs:

instance_region_1_az = "us-east-2a"
instance_region_2_az = "us-west-1b"
```

OK, so now you know how to deploy data sources and resources into different regions. What about modules? For example, in Chapter 3, you used Amazon RDS to deploy a single instance of a MySQL database in the staging environment (*stage/data-stores/mysql*):

```
provider "aws" {
  region = "us-east-2"
}

resource "aws_db_instance" "example" {
  identifier_prefix   = "terraform-up-and-running"
```

```
  engine              = "mysql"
  allocated_storage   = 10
  instance_class      = "db.t2.micro"
  skip_final_snapshot = true

  username = var.db_username
  password = var.db_password
}
```

This is fine in staging, but in production, a single database is a single point of failure. Fortunately, Amazon RDS natively supports *replication*, where your data is automatically copied from a primary database to a secondary database—a read-only *replica*—which is useful for scalability and as a standby in case the primary goes down. You can even run the replica in a totally different AWS region, so if one region goes down (e.g., there's a major outage in `us-east-2`), you can switch to the other region (e.g., `us-west-1`).

Let's turn that MySQL code in the staging environment into a reusable `mysql` module that supports replication. First, copy all the contents of *stage/data-stores/mysql*, which should include *main.tf*, *variables.tf*, and *outputs.tf*, into a new *modules/data-stores/mysql* folder. Next, open *modules/data-stores/mysql/variables.tf* and expose two new variables:

```
variable "backup_retention_period" {
  description = "Days to retain backups. Must be > 0 to enable replication."
  type        = number
  default     = null
}

variable "replicate_source_db" {
  description = "If specified, replicate the RDS database at the given ARN."
  type        = string
  default     = null
}
```

As you'll see shortly, you'll set the `backup_retention_period` variable on the primary database to enable replication, and you'll set the `replicate_source_db` variable on the secondary database to turn it into a replica. Open up *modules/data-stores/mysql/main.tf*, and update the `aws_db_instance` resource as follows:

1. Pass the `backup_retention_period` and `replicate_source_db` variables into parameters of the same name in the `aws_db_instance` resource.
2. If a database instance is a replica, AWS does not allow you to set the `engine`, `db_name`, `username`, or `password` parameters, as those

are all inherited from the primary. So you must add some conditional logic to the `aws_db_instance` resource to not set those parameters when the `replicate_source_db` variable is set.

Here's what the resource should look like after the changes:

```
resource "aws_db_instance" "example" {
  identifier_prefix   = "terraform-up-and-running"
  allocated_storage   = 10
  instance_class      = "db.t2.micro"
  skip_final_snapshot = true

  # Enable backups
  backup_retention_period = var.backup_retention_period

  # If specified, this DB will be a replica
  replicate_source_db = var.replicate_source_db

  # Only set these params if replicate_source_db is not set
  engine   = var.replicate_source_db == null ? "mysql" : null
  db_name  = var.replicate_source_db == null ? var.db_name : null
  username = var.replicate_source_db == null ? var.db_username : null
  password = var.replicate_source_db == null ? var.db_password : null
}
```

Note that for replicas, this implies that the `db_name`, `db_username`, and `db_password` input variables in this module should be optional, so it's a good idea to go back to *modules/data-stores/mysql/variables.tf* and set the `default` for those variables to `null`:

```
variable "db_name" {
  description = "Name for the DB."
  type        = string
  default     = null
}

variable "db_username" {
  description = "Username for the DB."
  type        = string
  sensitive   = true
  default     = null
}

variable "db_password" {
  description = "Password for the DB."
  type        = string
  sensitive   = true
  default     = null
}
```

To use the `replicate_source_db` variable, you'll need set it to the ARN of another RDS database, so you should also update *modules/data-stores/mysql/outputs.tf* to add the database ARN as an output variable:

```
output "arn" {
  value       = aws_db_instance.example.arn
  description = "The ARN of the database"
}
```

One more thing: you should add a `required_providers` block to this module to specify that this module expects to use the AWS Provider, and to specify which version of the provider the module expects.

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.0"
    }
  }
}
```

You'll see in a moment why this is important when working with multiple regions, too!

OK, you can now use this `mysql` module to deploy a MySQL primary and a MySQL replica in the production environment. First, create *live/prod/data-stores/mysql/variables.tf* to expose input variables for the database username and password (so you can pass these secrets in as environment variables, as discussed in Chapter 6):

```
variable "db_username" {
  description = "The username for the database"
  type        = string
  sensitive   = true
}

variable "db_password" {
  description = "The password for the database"
  type        = string
  sensitive   = true
}
```

Next, create *live/prod/data-stores/mysql/main.tf*, and use the `mysql` module to configure the primary as follows:

```
module "mysql_primary" {
  source = "../../../../modules/data-stores/mysql"

  db_name     = "prod_db"
  db_username = var.db_username
  db_password = var.db_password

  # Must be enabled to support replication
  backup_retention_period = 1
}
```

Now, add a second usage of the `mysql` module to create a replica:

```
module "mysql_replica" {
  source = "../../../../modules/data-stores/mysql"

  # Make this a replica of the primary
  replicate_source_db = module.mysql_primary.arn
}
```

Nice and short! All you're doing is passing the ARN of the primary database into the `replicate_source_db` parameter, which should spin up an RDS database as a replica.

There's just one problem: How do you tell the code to deploy the primary and replica into different regions? To do so, create two `provider` blocks, each with its own alias:

```
provider "aws" {
  region = "us-east-2"
  alias  = "primary"
}

provider "aws" {
  region = "us-west-1"
  alias  = "replica"
}
```

To tell a module which providers to use, you set the `providers` parameter. Here's how you configure the MySQL primary to use the `primary` provider (the one in `us-east-2`):

```
module "mysql_primary" {
  source = "../../../../modules/data-stores/mysql"

  providers = {
    aws = aws.primary
  }
```

```
  db_name     = "prod_db"
  db_username = var.db_username
  db_password = var.db_password

  # Must be enabled to support replication
  backup_retention_period = 1
}
```

And here is how you configure the MySQL replica to use the `replica` provider (the one in `us-west-1`):

```
module "mysql_replica" {
  source = "../../../../modules/data-stores/mysql"

  providers = {
    aws = aws.replica
  }

  # Make this a replica of the primary
  replicate_source_db = module.mysql_primary.arn
}
```

Notice that with modules, the `providers` (plural) parameter is a map, whereas with resources and data sources, the `provider` (singular) parameter is a single value. That's because each resource and data source deploys into exactly one provider, but a module may contain multiple data sources and resources and use multiple providers (you'll see an example of multiple providers in a module later). In the `providers` map you pass to a module, the key must match the local name of the provider in the `required_providers` map within the module (in this case, both are set to `aws`). This is yet another reason defining `required_providers` explicitly is a good idea in just about every module.

Alright, the last step is to create *live/prod/data-stores/mysql/outputs.tf* with the following output variables:

```
output "primary_address" {
  value       = module.mysql_primary.address
  description = "Connect to the primary database at this endpoint"
}

output "primary_port" {
  value       = module.mysql_primary.port
  description = "The port the primary database is listening on"
}

output "primary_arn" {
  value       = module.mysql_primary.arn
```

```
  description = "The ARN of the primary database"
}

output "replica_address" {
  value       = module.mysql_replica.address
  description = "Connect to the replica database at this endpoint"
}

output "replica_port" {
  value       = module.mysql_replica.port
  description = "The port the replica database is listening on"
}

output "replica_arn" {
  value       = module.mysql_replica.arn
  description = "The ARN of the replica database"
}
```

And now you're finally ready to deploy! Note that running `apply` to spin up a primary and replica can take a long time, some 20–30 minutes, so be patient:

```
$ terraform apply

(...)

Apply complete! Resources: 2 added, 0 changed, 0 destroyed.

Outputs:
primary_address = "terraform-up-and-running.cmyd6qwb.us-east-2.rds.amazonaws
primary_arn     = "arn:aws:rds:us-east-2:111111111111:db:terraform-up-and-ru
primary_port    = 3306
replica_address = "terraform-up-and-running.drctpdoe.us-west-1.rds.amazonaws
replica_arn     = "arn:aws:rds:us-west-1:111111111111:db:terraform-up-and-ru
replica_port    = 3306
```

And there you have it, cross-region replication! You can log into the RDS Console to confirm replication is working. As shown in Figure 7-2, you should see a primary in `us-east-2` and a replica in `us-west-1`.

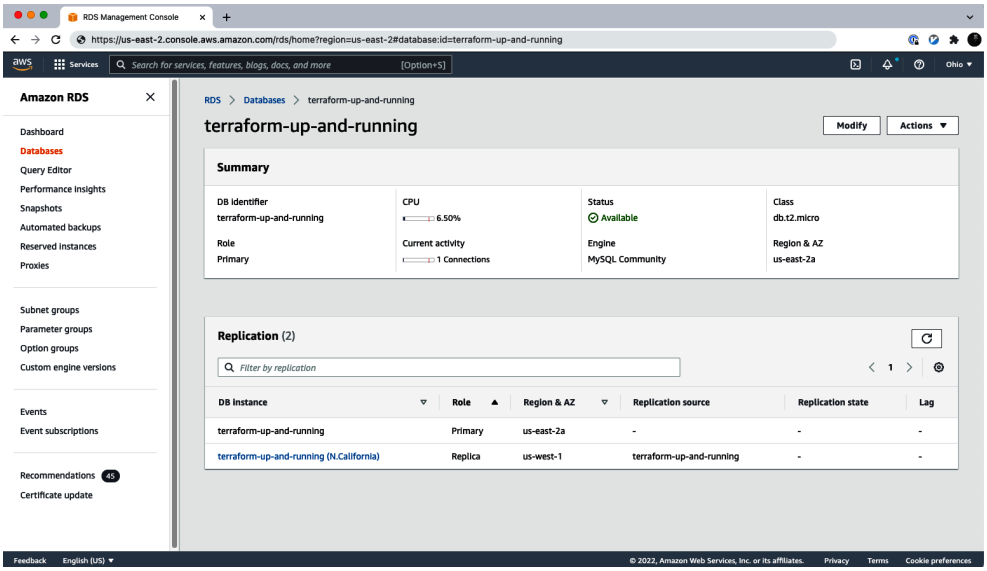

Figure 7-2. The RDS console shows a primary database in `us-east-2` and a replica in `us-west-1`.

As an exercise for the reader, I leave it up to you to update the staging environment (*stage/data-stores/mysql*) to use your `mysql` module (*modules/data-stores/mysql*) as well, but to configure it *without* replication, as you don't usually need that level of availability in pre-production environments.

As you can see in these examples, by using multiple providers with aliases, deploying resources across multiple regions with Terraform is pretty easy. However, I want to give two warnings before moving on:

Warning 1: Multiregion is hard

To run infrastructure in multiple regions around the world, especially in "active-active" mode, where more than one region is actively responding to user requests at the same time (as opposed to one region being a standby), there are many hard problems to solve, such as dealing with latency between regions, deciding between one writer (which means you have lower availability and higher latency) or multiple writers (which means you have either eventual consistency or sharding), figuring out how to generate unique IDs (the standard auto increment ID in most databases no longer suffices), working to meet local data regulations, and so on. These challenges are all beyond the scope of the book, but I figured I'd at least mention them to make it clear that multiregion deployments in the real world are not just a matter of tossing a few provider aliases into your Terraform code!

Warning 2: Use aliases sparingly

Although it's easy to use aliases with Terraform, I would caution against using them too often, *especially* when setting up multire-

gion infrastructure. One of the main reasons to set up multiregion infrastructure is so you can be resilient to the outage of one region: e.g., if `us-east-2` goes down, your infrastructure in `us-west-1` can keep running. But if you use a single Terraform module that uses aliases to deploy into both regions, then when one of those regions is down, the module will not be able to connect to that region, and any attempt to run `plan` or `apply` will fail. So right when you need to roll out changes—when there's a major outage—your Terraform code will stop working.

More generally, as discussed in Chapter 3, you should keep environments completely isolated: so instead of managing multiple regions in one module with aliases, you manage each region in separate modules. That way, you minimize the blast radius, both from your own mistakes (e.g., if you accidentally break something in one region, it's less likely to affect the other) and from problems in the world itself (e.g., an outage in one region is less likely to affect the other).

So when does it make sense to use aliases? Typically, aliases are a good fit when the infrastructure you're deploying across several aliased providers is truly coupled and you want to always deploy it together. For example, if you wanted to use Amazon CloudFront as a CDN (Content Distribution Network), and to provision a TLS certificate for it using AWS Certification Manager (ACM), then AWS requires the certificate to be created in the `us-east-1` region, no matter what other regions you happen to be using for CloudFront itself. In that case, your code may have two `provider` blocks, one for the primary region you want to use for CloudFront and one with an `alias` hardcoded specifically to `us-east-1` for configuring the TLS certificate. Another use case for aliases is if you're deploying resources designed for use across many regions: for example, AWS recommends deploying GuardDuty, an automated threat detection service, in every single region you're using in your AWS account. In this case, it may make sense to have a module with a `provider` block and custom `alias` for each AWS region.

Beyond a few corner cases like this, using aliases to handle multiple regions is relatively rare. A more common use case for aliases is when you have multiple providers that need to authenticate in different ways, such as each one authenticating to a different AWS account.

Working with Multiple AWS Accounts

So far, throughout this book, you've likely been using a single AWS account for all of your infrastructure. For production code, it's more com-

mon to use multiple AWS accounts: e.g., you put your staging environment in a stage account, your production environment in a prod account, and so on. This concept applies to other clouds too, such as Azure and Google Cloud. Note that I'll be using the term *account* in this book, even though some clouds use slightly different terminology for the same concept (e.g., Google Cloud calls them *projects* instead of accounts).

The main reasons for using multiple accounts are as follows:

Isolation (aka compartmentalization)

You use separate accounts to isolate different environments from each other and to limit the "blast radius" when things go wrong. For example, putting your staging and production environments in separate accounts ensures that if an attacker manages to break into staging, they still have no access whatsoever to production. Likewise, this isolation ensures that a developer making changes in staging is less likely to accidentally break something in production.

Authentication and authorization

If everything is in one account, it's tricky to grant access to some things (e.g., the staging environment) but not accidentally grant access to other things (e.g., the production environment). Using multiple accounts makes it easier to have fine-grained control, as any permissions you grant in one account have no effect on any other account.

The authentication requirements of multiple accounts also help reduce the chance of mistakes. With everything in a single account, it's too easy to make the mistake where you think you're making a change in, say, your staging environment, but you're actually making the change in production (which can be a disaster if the change you're making is, for example, to drop all database tables). With multiple accounts, this is less likely, as authenticating to each account requires a separate set of steps.

Note that having multiple accounts does *not* imply that developers have multiple separate user profiles (e.g., a separate IAM user in each AWS account). In fact, that would be an antipattern, as that would require managing multiple sets of credentials, permissions, etc. Instead, you can configure just about all the major clouds so that each developer has exactly one user profile, which they can use to authenticate to any account they have access to. The cross-account authentication mechanism varies depending on the cloud you're using: e.g., in AWS, you can authenticate across AWS accounts by assuming IAM roles, as you'll see shortly.

Auditing and reporting

> A properly configured account structure will allow you to maintain an audit trail of all the changes happening in all your environments, check if you're adhering to compliance requirements, and detect anomalies. Moreover, you'll be able to have consolidated billing, with all the charges for all of your accounts in one place, including cost breakdowns by account, service, tag, etc. This is especially useful in large organizations, as it allows finance to track and budget spending by team simply by looking at which account the charges are coming from.

Let's go through a multi-account example with AWS. First, you'll want to create a new AWS account to use for testing. Since you already have one AWS account, to create new *child accounts*, you can use AWS Organizations, which ensures that the billing from all the child accounts rolls up into the parent account (sometimes called the *root account*) and gives you a dashboard you can use to manage all the child accounts.

Head over to the AWS Organizations Console, and click the "Add an AWS account" button, as shown in Figure 7-3.

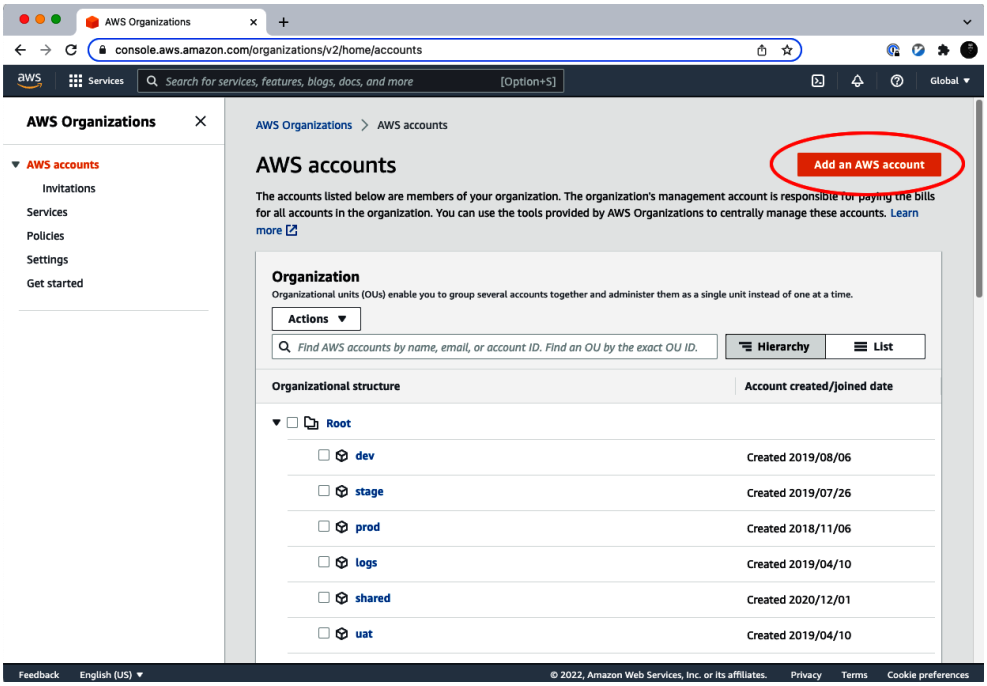

Figure 7-3. Use AWS Organizations to create a new AWS account.

On the next page, fill in the following info, as shown in Figure 7-4:

AWS account name

> The name to use for the account. For example, if this account was going to be used for your staging environment, you might name it "staging."

Email address of the account's owner

The email address to use for the root user of the AWS account. Note that every AWS account must use a different email address for the root user, so you can't reuse the email address you used to create your first (root) AWS account (see "How to Get Multiple Aliases from One Email Address" for a workaround). So what about the root user's password? By default, AWS does not configure a password for the root user of a new child account (you'll see shortly an alternative way to authenticate to the child account). If you ever do want to log in as this root user, after you create the child account, you'll need to go through the password reset flow with the email address you're specifying here.

IAM role name

When AWS Organizations creates a child AWS account, it automatically creates an IAM role within that child AWS account that has admin permissions and can be assumed from the parent account. This is convenient, as it allows you to authenticate to the child AWS account without having to create any IAM users or IAM roles yourself. I recommend leaving this IAM role name at the default value of `OrganizationAccountAccessRole`.

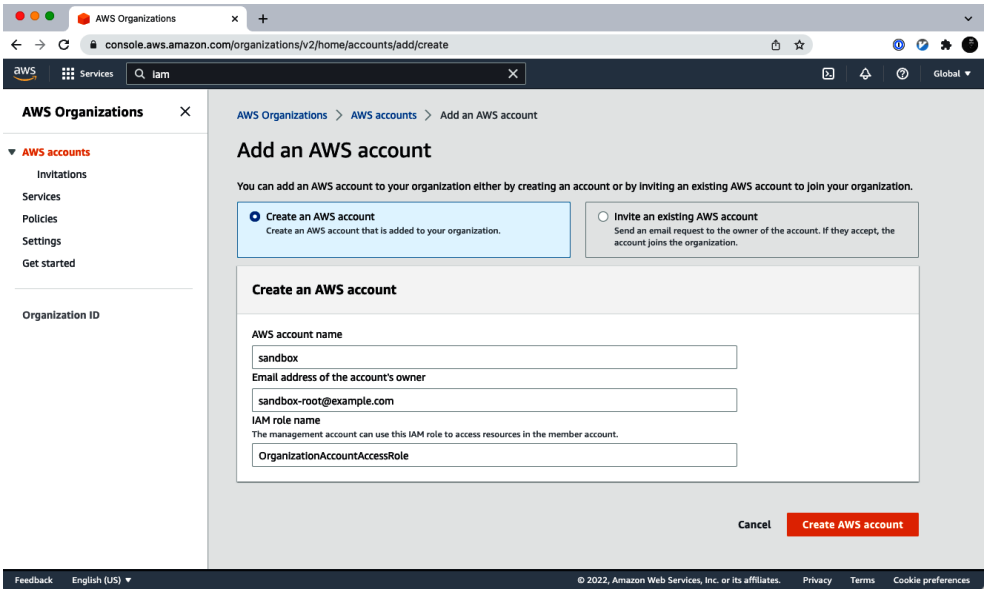

Figure 7-4. Fill in the details for the new AWS account.

HOW TO GET MULTIPLE ALIASES FROM ONE EMAIL ADDRESS

If you use Gmail, you can get multiple email aliases out of a single address by taking advantage of the fact that Gmail ignores everything after a plus sign in an email address. For example, if your Gmail address is *example@gmail.com*, you can send email to *example+foo@gmail.com* and *example+any-text-you-want@gmail.com*, and all of those emails will go to *example@gmail.com*. This also works if your company uses Gmail via Google Workspace, even with a custom domain: e.g., *example+dev@company.com* and *example+stage@company.com* will all go to *example@company.com*.

This is useful if you're creating a dozen child AWS accounts, as instead of having to create a dozen totally separate email addresses, you could use *example+dev@company.com* for your dev account, *example+stage@company.com* for your stage account, and so on; AWS will see each of those email addresses as a different, unique address, but under the hood, all the emails will go to the same account.

Click the Create AWS Account button, wait a few minutes for AWS to create the account, and then jot down the 12-digit ID of the AWS account that gets created. For the rest of this chapter, let's assume the following:

- Parent AWS account ID: `111111111111`
- Child AWS account ID: `222222222222`

You can authenticate to your new child account from the AWS Console by clicking your username and selecting "Switch role", as shown in Figure 7-5.

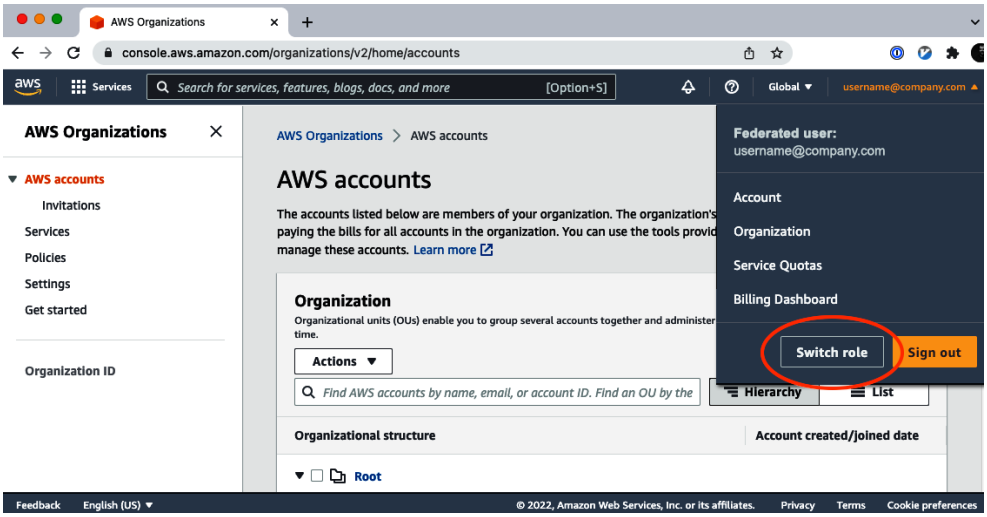

Figure 7-5. Select the "Switch role" button.

Next, enter the details for the IAM role you want to assume, as shown in Figure 7-6:

Account

The 12-digit ID of the AWS account to switch to. You'll want to enter the ID of your new child account.

Role

The name of the IAM role to assume in that AWS account. Enter the name you used for the IAM role when creating the new child account, which is `OrganizationAccountAccessRole` by default.

Display name

AWS will create a shortcut in the nav to allow you to switch to this account in the future with a single click. This is the name to show in this shortcut. It only affects your IAM user in this browser.

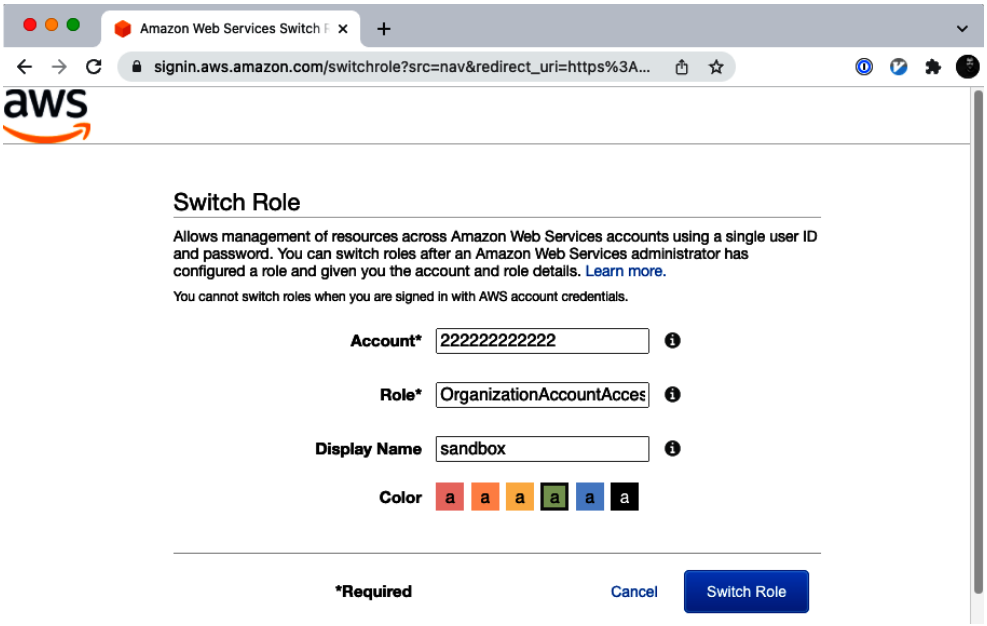

Figure 7-6. Enter the details for the role to switch to.

Click Switch Role and, voilà, AWS should log you into the web console of the new AWS account!

Let's now write an example Terraform module in *examples/multi-account-root* that can authenticate to multiple AWS accounts. Just as with the multiregion AWS example, you will need to add two `provider` blocks in *main.tf*, each with a different alias. First, the `provider` block for the parent AWS account:

```
provider "aws" {
  region = "us-east-2"
  alias  = "parent"
}
```

Next, the `provider` block for the child AWS account:

```
provider "aws" {
  region = "us-east-2"
  alias  = "child"
}
```

To be able to authenticate to the child AWS account, you'll assume an IAM role. In the web console, you did this by clicking the Switch Role button; in your Terraform code, you do this by adding an `assume_role` block to the child `provider` block:

```
provider "aws" {
  region = "us-east-2"
  alias  = "child"

  assume_role {
    role_arn = "arn:aws:iam::<ACCOUNT_ID>:role/<ROLE_NAME>"
  }
}
```

In the `role_arn` parameter, you'll need to replace `ACCOUNT_ID` with your child account ID and `ROLE_NAME` with the name of the IAM role in that account, just as you did when switching roles in the web console. Here's what it looks like with the account ID `222222222222` and role name `OrganizationAccountAccessRole` plugged in:

```
provider "aws" {
  region = "us-east-2"
  alias  = "child"

  assume_role {
    role_arn = "arn:aws:iam::222222222222:role/OrganizationAccountAccessRole
  }
}
```

Now, to check this is actually working, add two `aws_caller_identity` data sources, and configure each one to use a different provider:

```
data "aws_caller_identity" "parent" {
  provider = aws.parent
}

data "aws_caller_identity" "child" {
  provider = aws.child
}
```

Finally, add output variables in *outputs.tf* to print out the account IDs:

```
output "parent_account_id" {
  value       = data.aws_caller_identity.parent.account_id
  description = "The ID of the parent AWS account"
}

output "child_account_id" {
  value       = data.aws_caller_identity.child.account_id
  description = "The ID of the child AWS account"
}
```

Run `apply`, and you should see the different IDs for each account:

```
$ terraform apply

(...)

Apply complete! Resources: 0 added, 0 changed, 0 destroyed.

Outputs:

parent_account_id = "111111111111"
child_account_id = "222222222222"
```

And there you have it: by using provider aliases and `assume_role` blocks, you now know how to write Terraform code that can operate across multiple AWS accounts.

As with the multiregion section, a few warnings:

Warning 1: Cross-account IAM roles are double opt-in

In order for an IAM role to allow access from one AWS account to another—e.g., to allow an IAM role in account `222222222222` to be assumed from account `111111111111`—you need to grant permissions in *both* AWS accounts:

- First, in the AWS account where the IAM role lives (e.g., the child account `222222222222`), you must configure its assume role policy to trust the other AWS account (e.g., the parent account `111111111111`). This happened magically for you with the `OrganizationAccountAccessRole` IAM role because AWS Organizations automatically configures the assume role policy of this IAM role to trust the parent account. However, for any custom IAM roles you create, you need to remember to explicitly grant the `sts:AssumeRole` permission yourself.
- Second, in the AWS account from which you assume the role (e.g., the parent account `111111111111`), you must *also* grant your user permissions to assume that IAM role. Again, this hap-

pened for you magically because, in Chapter 2, you gave your IAM user `AdministratorAccess`, which gives you permissions to do just about everything in the parent AWS account, including assuming IAM roles. In most real-world use cases, your user won't be (shouldn't be!) an admin, so you'll need to explicitly grant your user `sts:AssumeRole` permissions on the IAM role(s) you want to be able to assume.

Warning 2: Use aliases sparingly

I said this in the multiregion example, but it bears repeating: although it's easy to use aliases with Terraform, I would caution against using them too often, including with multi-account code. Typically, you use multiple accounts to create separation between them, so if something goes wrong in one account, it doesn't affect the other. Modules that deploy across multiple accounts go against this principle. Only do it when you *intentionally* want to have resources in multiple accounts coupled and deployed together.

Creating Modules That Can Work with Multiple Providers

When working with Terraform modules, you typically work with two types of modules:

Reusable modules

These are low-level modules that are not meant to be deployed directly but instead are to be combined with other modules, resources, and data sources.

Root modules

These are high-level modules that combine multiple reusable modules into a single unit that is meant to be deployed directly by running `apply` (in fact, the definition of a root module is it's the one on which you run `apply`).

The multiprovider examples you've seen so far have put all the `provider` blocks into the root module. What do you do if you want to create a reusable module that works with multiple providers? For example, what if you wanted to turn the multi-account code from the previous section into a reusable module? As a first step, you might put all that code, unchanged, into the *modules/multi-account* folder. Then, you could create a new example to test it with in the *examples/multi-account-module* folder, with a *main.tf* that looks like this:

```
module "multi_account_example" {
  source = "../../modules/multi-account"
}
```

If you run `apply` on this code, it'll work, but there is a problem: all of the `provider` configuration is now hidden in the module itself (in *modules/multi-account*). Defining `provider` blocks within reusable modules is an antipattern for several reasons:

Configuration problems

If you have `provider` blocks defined in your reusable module, then that module controls all the configuration for that `provider`. For example, the IAM role ARN and regions to use are currently hardcoded in the *modules/multi-account* module. You could, of course, expose input variables to allow users to set the regions and IAM role ARNs, but that's only the tip of the iceberg. If you browse the AWS Provider documentation, you'll find that there are roughly *50* different configuration options you can pass into it! Many of these parameters are going to be important for users of your module, as they control how to authenticate to AWS, what region to use, what account (or IAM role) to use, what endpoints to use when talking to AWS, what tags to apply or ignore, and much more. Having to expose 50 extra variables in a module will make that module very cumbersome to maintain and use.

Duplication problems

Even if you expose those 50 settings in your module, or whatever subset you believe is important, you're creating code duplication for users of your module. That's because it's common to combine multiple modules together, and if you have to pass in some subset of 50 settings into each of those modules in order to get them to all authenticate correctly, you're going to have to copy and paste a lot of parameters, which is tedious and error prone.

Performance problems

Every time you include a `provider` block in your code, Terraform spins up a new process to run that provider, and communicates with that process via RPC. If you have a handful of `provider` blocks, this works just fine, but as you scale up, it may cause performance problems. Here's a real-world example: a few years ago, I created reusable modules for CloudTrail, AWS Config, GuardDuty, IAM Access Analyzer, and Macie. Each of these AWS services is supposed to be deployed into every region in your AWS account, and as AWS had ~25 regions, I included 25 `provider` blocks in each of

these modules. I then created a single root module to deploy all of these as a "baseline" in my AWS accounts: if you do the math, that's 5 modules with 25 `provider` blocks each, or 125 `provider` blocks total. When I ran `apply`, Terraform would fire up 125 processes, each making hundreds of API and RPC calls. With thousands of concurrent network requests, my CPU would start thrashing, and a single `plan` could take 20 minutes. Worse yet, this would sometimes overload the network stack, leading to intermittent failures in API calls, and `apply` would fail with sporadic errors.

Therefore, as a best practice, you should *not* define any `provider` blocks in your reusable modules and instead allow your users to create the `provider` blocks they need solely in their root modules. But then, how do you build a module that can work with multiple providers? If the module has no `provider` blocks in it, how do you define provider aliases that you can reference in your resources and data sources?

The solution is to use *configuration aliases*. These are very similar to the provider aliases you've seen already, except they aren't defined in a `provider` block. Instead, you define them in a `required_providers` block.

Open up *modules/multi-account/main.tf*, remove the nested `provider` blocks, and replace them with a `required_providers` block with configuration aliases as follows:

```
terraform {
  required_providers {
    aws = {
      source                = "hashicorp/aws"
      version               = "~> 4.0"
      configuration_aliases = [aws.parent, aws.child]
    }
  }
}
```

Just as with normal provider aliases, you can pass configuration aliases into resources and data sources using the `provider` parameter:

```
data "aws_caller_identity" "parent" {
  provider = aws.parent
}

data "aws_caller_identity" "child" {
  provider = aws.child
}
```

The key difference from normal provider aliases is that configuration aliases don't create any providers themselves; instead, they force users of your module to explicitly pass in a provider for each of your configuration aliases using a `providers` map.

Open up *examples/multi-account-module/main.tf*, and define the `provider` blocks as before:

```
provider "aws" {
  region = "us-east-2"
  alias  = "parent"
}

provider "aws" {
  region = "us-east-2"
  alias  = "child"

  assume_role {
    role_arn = "arn:aws:iam::222222222222:role/OrganizationAccountAccessRole
  }
}
```

And now you can pass them into the *modules/multi-account* module as follows:

```
module "multi_account_example" {
  source = "../../modules/multi-account"

  providers = {
    aws.parent = aws.parent
    aws.child  = aws.child
  }
}
```

The keys in the `providers` map must match the names of the configuration aliases within the module; if any of the names from configuration aliases are missing in the `providers` map, Terraform will show an error. This way, when you're building a reusable module, you can define what providers that module needs, and Terraform will ensure users pass those providers in; and when you're building a root module, you can define your `provider` blocks just once and pass around references to them to the reusable modules you depend on.

Working with Multiple Different

Providers

You've now seen how to work with multiple providers when all of them are the same type of provider: e.g., multiple copies of the `aws` provider. This section talks about how to work with multiple different providers.

Readers of the first two editions of this book often asked for examples of using multiple clouds together (*multicloud*), but I couldn't find much useful to share. In part, this is because using multiple clouds is usually a bad practice,[2] but even if you're forced to do it (most large companies are multicloud, whether they want to be or not), it's rare to manage multiple clouds in a single module for the same reason it's rare to manage multiple regions or accounts in a single module. If you're using multiple clouds, you're far better off managing each one in a separate module.

Moreover, translating every single AWS example in the book into the equivalent solutions for other clouds (Azure and Google Cloud) is impractical: the book would end up way too long, and while you would learn more about each cloud, you wouldn't learn any new Terraform concepts along the way, which is the real goal of the book. If you do want to see examples of what the Terraform code for similar infrastructure looks like across different clouds, have a look at the *examples* folder in the Terratest repo. As you'll see in Chapter 9, Terratest provides a set of tools for writing automated tests for different types of infrastructure code and different types of clouds, so in the *examples* folder you'll find Terraform code for similar infrastructure in AWS, Google Cloud, and Azure, including individual servers, groups of servers, databases, and more. You'll also find automated tests for all those examples in the *test* folder.

In this book, instead of an unrealistic multicloud example, I decided to instead show you how to use multiple providers together in a slightly more realistic scenario (one that was also requested by many readers of the first two editions): namely, how to use the AWS Provider with the Kubernetes provider to deploy Dockerized apps. Kubernetes is, in many ways, a cloud of its own—it can run applications, networks, data stores, load balancers, secret stores, and much more—so, in a sense, this is both a multiprovider and multicloud example. And because Kubernetes is a cloud, that means there is a lot to learn, so I'm going to have to build up to it one step at a time, starting with mini crash courses on Docker and Kubernetes, before finally moving on to the full multiprovider example that uses both AWS and Kubernetes:

- A crash course on Docker
- A crash course on Kubernetes

- Deploying Docker containers in AWS using Elastic Kubernetes Service (EKS)

A Crash Course on Docker

As you may remember from Chapter 1, Docker images are like self-contained "snapshots" of the operating system (OS), the software, the files, and all other relevant details. Let's now see Docker in action.

First, if you don't have Docker installed already, follow the instructions on the Docker website to install Docker Desktop for your operating system. Once it's installed, you should have the `docker` command available on your command line. You can use the `docker run` command to run Docker images locally:

```
$ docker run <IMAGE> [COMMAND]
```

where `IMAGE` is the Docker image to run and `COMMAND` is an optional command to execute. For example, here's how you can run a Bash shell in an Ubuntu 20.04 Docker image (note that the following command includes the `-it` flag so you get an interactive shell where you can type):

```
$ docker run -it ubuntu:20.04 bash

Unable to find image 'ubuntu:20.04' locally
20.04: Pulling from library/ubuntu
Digest: sha256:669e010b58baf5beb2836b253c1fd5768333f0d1dbcb834f7c07a4dc93f47
Status: Downloaded newer image for ubuntu:20.04

root@d96ad3779966:/#
```

And voilà, you're now in Ubuntu! If you've never used Docker before, this can seem fairly magical. Try running some commands. For example, you can look at the contents of */etc/os-release* to verify you really are in Ubuntu:

```
root@d96ad3779966:/# cat /etc/os-release
NAME="Ubuntu"
VERSION="20.04.3 LTS (Focal Fossa)"
ID=ubuntu
ID_LIKE=debian
PRETTY_NAME="Ubuntu 20.04.3 LTS"
VERSION_ID="20.04"
VERSION_CODENAME=focal
```

How did this happen? Well, first, Docker searches your local filesystem for the `ubuntu:20.04` image. If you don't have that image downloaded already, Docker downloads it automatically from Docker Hub, which is a *Docker Registry* that contains shared Docker images. The `ubuntu:20.04` image happens to be a public Docker image—an official one maintained by the Docker team—so you're able to download it without any authentication. However, it's also possible to create private Docker images that only certain authenticated users can use.

Once the image is downloaded, Docker runs the image, executing the `bash` command, which starts an interactive Bash prompt, where you can type. Try running the `ls` command to see the list of files:

```
root@d96ad3779966:/# ls -al
total 56
drwxr-xr-x   1 root root 4096 Feb 22 14:22 .
drwxr-xr-x   1 root root 4096 Feb 22 14:22 ..
lrwxrwxrwx   1 root root    7 Jan 13 16:59 bin -> usr/bin
drwxr-xr-x   2 root root 4096 Apr 15  2020 boot
drwxr-xr-x   5 root root  360 Feb 22 14:22 dev
drwxr-xr-x   1 root root 4096 Feb 22 14:22 etc
drwxr-xr-x   2 root root 4096 Apr 15  2020 home
lrwxrwxrwx   1 root root    7 Jan 13 16:59 lib -> usr/lib
drwxr-xr-x   2 root root 4096 Jan 13 16:59 media
(...)
```

You might notice that's not your filesystem. That's because Docker images run in containers that are isolated at the userspace level: when you're in a container, you can only see the filesystem, memory, networking, etc., in that container. Any data in other containers, or on the underlying host operating system, is not accessible to you, and any data in your container is not visible to those other containers or the underlying host operating system. This is one of the things that makes Docker useful for running applications: the image format is self-contained, so Docker images run the same way no matter where you run them, and no matter what else is running there.

To see this in action, write some text to a *test.txt* file as follows:

```
root@d96ad3779966:/# echo "Hello, World!" > test.txt
```

Next, exit the container by hitting Ctrl-D on Windows and Linux or Cmd-D on macOS, and you should be back in your original command prompt on your underlying host OS. If you try to look for the *test.txt* file you just wrote, you'll see that it doesn't exist: the container's filesystem is totally isolated from your host OS.

Now, try running the same Docker image again:

```
$ docker run -it ubuntu:20.04 bash
root@3e0081565a5d:/#
```

Notice that this time, since the `ubuntu:20.04` image is already downloaded, the container starts almost instantly. This is another reason Docker is useful for running applications: unlike virtual machines, containers are lightweight, boot up quickly, and incur little CPU or memory overhead.

You may also notice that the second time you fired up the container, the command prompt looked different. That's because you're now in a totally new container; any data you wrote in the previous one is no longer accessible to you. Run `ls -al` and you'll see that the *test.txt* file does not exist. Containers are isolated not only from the host OS but also from each other.

Hit Ctrl-D or Cmd-D again to exit the container, and back on your host OS, run the `docker ps -a` command:

```
$ docker ps -a
CONTAINER ID   IMAGE             COMMAND    CREATED          STATUS
3e0081565a5d   ubuntu:20.04      "bash"     5 min ago    Exited (0) 16 sec ag
d96ad3779966   ubuntu:20.04      "bash"     14 min ago   Exited (0) 5 min ago
```

This will show you all the containers on your system, including the stopped ones (the ones you exited). You can start a stopped container again by using the `docker start <ID>` command, setting `ID` to an ID from the `CONTAINER ID` column of the `docker ps` output. For example, here is how you can start the first container up again (and attach an interactive prompt to it via the `-ia` flags):

```
$ docker start -ia d96ad3779966
root@d96ad3779966:/#
```

You can confirm this is really the first container by outputting the contents of *test.txt*:

```
root@d96ad3779966:/# cat test.txt
Hello, World!
```

Let's now see how a container can be used to run a web app. Hit Ctrl-D or Cmd-D again to exit the container, and back on your host OS, run a new container:

```
$ docker run training/webapp
 * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
```

The training/webapp image contains a simple Python "Hello, World" web app for testing. When you run the image, it fires up the web app, listening on port 5000 by default. However, if you open a new terminal on your host operating system and try to access the web app, it won't work:

```
$ curl localhost:5000
curl: (7) Failed to connect to localhost port 5000: Connection refused
```

What's the problem? Actually, it's not a problem but a feature! Docker containers are isolated from the host operating system and other containers, not only at the filesystem level but also in terms of networking. So while the container really is listening on port 5000, that is only on a port *inside* the container, which isn't accessible on the host OS. If you want to expose a port from the container on the host OS, you have to do it via the `-p` flag.

First, hit Ctrl-C to shut down the `training/webapp` container: note that it's C this time, not D, and it's Ctrl regardless of OS, as you're shutting down a process, rather than exiting an interactive prompt. Now rerun the container but this time with the `-p` flag as follows:

```
$ docker run -p 5000:5000 training/webapp
 * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
```

Adding `-p 5000:5000` to the command tells Docker to expose port 5000 inside the container on port 5000 of the host OS. In another terminal on your host OS, you should now be able to see the web app working:

```
$ curl localhost:5000
Hello world!
```

CLEANING UP CONTAINERS

Every time you run `docker run` and exit, you are leaving behind containers, which take up disk space. You may wish to clean them up with the `docker rm <CONTAINER_ID>` command, where `CONTAINER_ID` is the ID of the container from the `docker ps` output. Alternatively, you could include the `--rm` flag in your `docker run` command to have Docker automatically clean up when you exit the container.

A Crash Course on Kubernetes

Kubernetes is an orchestration tool for Docker, which means it's a platform for running and managing Docker containers on your servers, including scheduling (picking which servers should run a given container workload), auto healing (automatically redeploying containers that failed), auto scaling (scaling the number of containers up and down in response to load), load balancing (distributing traffic across containers), and much more.

Under the hood, Kubernetes consists of two main pieces:

Control plane

The control plane is responsible for managing the Kubernetes cluster. It is the "brains" of the operation, responsible for storing the state of the cluster, monitoring containers, and coordinating actions across the cluster. It also runs the API server, which provides an API you can use from command-line tools (e.g., `kubectl`), web UIs (e.g., the Kubernetes Dashboard), and IaC tools (e.g., Terraform) to control what's happening in the cluster.

Worker nodes

The worker nodes are the servers used to actually run your containers. The worker nodes are entirely managed by the control plane, which tells each worker node what containers it should run.

Kubernetes is open source, and one of its strengths is that you can run it anywhere: in any public cloud (e.g., AWS, Azure, Google Cloud), in your own datacenter, and even on your own developer workstation. A little later in this chapter, I'll show you how you can run Kubernetes in the cloud (in AWS), but for now, let's start small and run it locally. This is easy to do if you installed a relatively recent version of Docker Desktop, as it has the ability to fire up a Kubernetes cluster locally with just a few clicks.

If you open Docker Desktop's preferences on your computer, you should see Kubernetes in the nav, as shown in Figure 7-7.

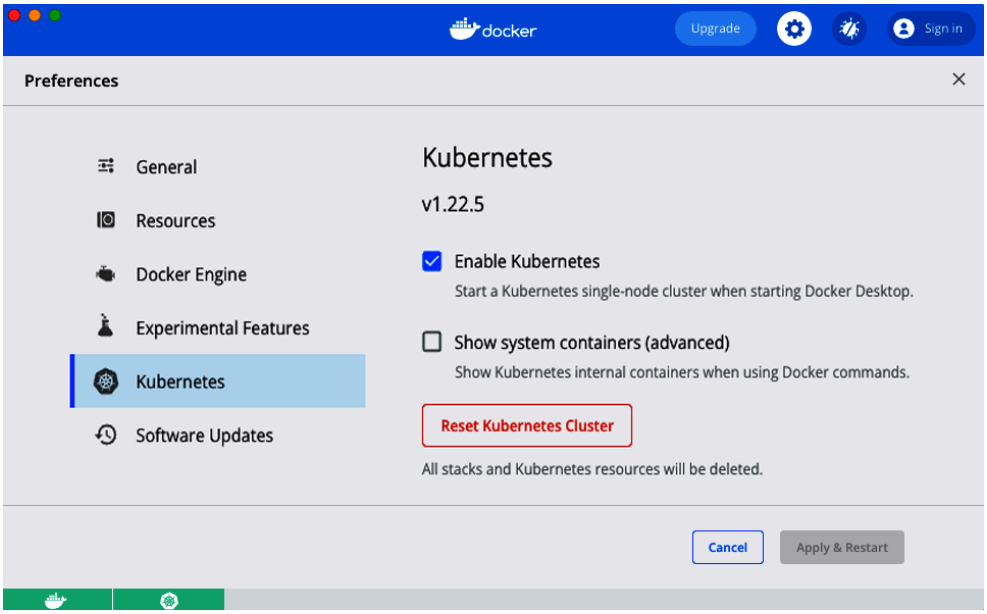

Figure 7-7. Enable Kubernetes on Docker Desktop.

If it's not enabled already, check the Enable Kubernetes checkbox, click Apply & Restart, and wait a few minutes for that to complete. In the meantime, follow the instructions on the Kubernetes website to install `kubectl`, which is the command-line tool for interacting with Kubernetes.

To use `kubectl`, you must first update its configuration file, which lives in *$HOME/.kube/config* (that is, the *.kube* folder of your home directory), to tell it what Kubernetes cluster to connect to. Conveniently, when you enable Kubernetes in Docker Desktop, it updates this config file for you, adding a `docker-desktop` entry to it, so all you need to do is tell `kubectl` to use this configuration as follows:

```
$ kubectl config use-context docker-desktop
Switched to context "docker-desktop".
```

Now you can check if your Kubernetes cluster is working with the `get nodes` command:

```
$ kubectl get nodes
NAME             STATUS   ROLES                  AGE   VERSION
docker-desktop   Ready    control-plane,master   95m   v1.22.5
```

The `get nodes` command shows you information about all the nodes in your cluster. Since you're running Kubernetes locally, your computer is the only node, and it's running both the control plane and acting as a worker node. You're now ready to run some Docker containers!

To deploy something in Kubernetes, you create Kubernetes *objects*, which are persistent entities you write to the Kubernetes cluster (via the API server) that record your intent: e.g., your intent to have specific Docker images running. The cluster runs a *reconciliation loop*, which continuously checks the objects you stored in it and works to make the state of the cluster match your intent.

There are many different types of Kubernetes objects available. For the examples in this book, let's use the following two objects:

Kubernetes Deployment

A *Kubernetes Deployment* is a declarative way to manage an application in Kubernetes. You declare what Docker images to run, how many copies of them to run (called *replicas*), a variety of settings for those images (e.g., CPU, memory, port numbers, environment variables), and the strategy to roll out updates to those images, and the Kubernetes Deployment will then work to ensure that the requirements you declared are always met. For example, if you specified you wanted three replicas, but one of the worker nodes went down so only two replicas are left, the Deployment will automatically spin up a third replica on one of the other worker nodes.

Kubernetes Service

A *Kubernetes Service* is a way to expose a web app running in Kubernetes as a networked service. For example, you can use a Kubernetes Service to configure a load balancer that exposes a public endpoint and distributes traffic from that endpoint across the replicas in a Kubernetes Deployment.

The idiomatic way to interact with Kubernetes is to create YAML files describing what you want—e.g., one YAML file that defines the Kubernetes Deployment and another one that defines the Kubernetes Service—and to use the `kubectl apply` command to submit those objects to the cluster. However, using raw YAML has drawbacks, such as a lack of support for code reuse (e.g., variables, modules), abstraction (e.g., loops, if-statements), clear standards on how to store and manage the YAML files (e.g., to track changes to the cluster over time), and so on. Therefore, many Kubernetes users turn to alternatives, such as Helm or Terraform. Since this is a book on Terraform, I'm going to show you how to create a Terraform module called `k8s-app` (K8S is an acronym for Kubernetes in the same way that I18N is an acronym for internationalization) that deploys an app in Kubernetes using a Kubernetes Deployment and Kubernetes Service.

Create a new module in the *modules/services/k8s-app* folder. Within that folder, create a *variables.tf* file that defines the module's API via the following input variables:

```
variable "name" {
  description = "The name to use for all resources created by this module"
  type        = string
}

variable "image" {
  description = "The Docker image to run"
  type        = string
}

variable "container_port" {
  description = "The port the Docker image listens on"
  type        = number
}

variable "replicas" {
  description = "How many replicas to run"
  type        = number
}

variable "environment_variables" {
  description = "Environment variables to set for the app"
  type        = map(string)
  default     = {}
}
```

This should give you just about all the inputs you need for creating the Kubernetes Deployment and Service. Next, add a *main.tf* file, and at the top, add the `required_providers` block to it with the Kubernetes provider:

```
terraform {
  required_version = ">= 1.0.0, < 2.0.0"

  required_providers {
    kubernetes = {
      source  = "hashicorp/kubernetes"
      version = "~> 2.0"
    }
  }
}
```

Hey, a new provider, neat! OK, let's make use of that provider to create a Kubernetes Deployment by using the `kubernetes_deployment` resource:

```
resource "kubernetes_deployment" "app" {
}
```

There are quite a few settings to configure within the `kubernetes_deployment` resource, so let's go through them one at a time. First, you need to configure the `metadata` block:

```
resource "kubernetes_deployment" "app" {
  metadata {
    name = var.name
  }
}
```

Every Kubernetes object includes metadata that can be used to identify and target that object in API calls. In the preceding code, I'm setting the Deployment name to the `name` input variable.

The rest of the configuration for the `kubernetes_deployment` resource goes into the `spec` block:

```
resource "kubernetes_deployment" "app" {
  metadata {
    name = var.name
  }

  spec {
  }
}
```

The first item to put into the `spec` block is to specify the number of replicas to create:

```
  spec {
    replicas = var.replicas
  }
```

Next, define the `template` block:

```
  spec {
    replicas = var.replicas

    template {
    }
  }
```

In Kubernetes, instead of deploying one container at a time, you deploy *Pods*, which are groups of containers that are meant to be deployed together. For example, you could have a Pod with one container to run a web app (e.g., the Python app you saw earlier) and another container that gathers metrics on the web app and sends them to a central service (e.g., Datadog). The `template` block is where you define the *Pod Template*, which specifies what container(s) to run, the ports to use, environment variables to set, and so on.

One important ingredient in the Pod Template will be the labels to apply to the Pod. You'll need to reuse these labels in several places—e.g., the Kubernetes Service uses labels to identify the Pods that need to be load balanced—so let's define those labels in a local variable called `pod_labels`:

```
locals {
  pod_labels = {
    app = var.name
  }
}
```

And now use `pod_labels` in the `metadata` block of the Pod Template:

```
  spec {
    replicas = var.replicas

    template {
      metadata {
        labels = local.pod_labels
      }
    }
  }
```

Next, add a `spec` block inside of `template`:

```
  spec {
    replicas = var.replicas

    template {
      metadata {
        labels = local.pod_labels
      }

      spec {
        container {
          name  = var.name
          image = var.image
```

```
            port {
              container_port = var.container_port
            }

            dynamic "env" {
              for_each = var.environment_variables
              content {
                name  = env.key
                value = env.value
              }
            }
          }
        }
      }
    }
```

There's a lot here, so let's go through it one piece at a time:

container

Inside the `spec` block, you can define one or more `container` blocks to specify which Docker containers to run in this Pod. To keep this example simple, there's just one `container` block in the Pod. The rest of these items are all within this `container` block.

name

The name to use for the container. I've set this to the `name` input variable.

image

The Docker image to run in the container. I've set this to the `image` input variable.

port

The ports to expose in the container. To keep the code simple, I'm assuming the container only needs to listen on one port, set to the `container_port` input variable.

env

The environment variables to expose to the container. I'm using a `dynamic` block with `for_each` (two concepts you may remember from Chapter 5) to set this to the variables in the `environment_variables` input variable.

OK, that wraps up the Pod Template. There's just one thing left to add to the `kubernetes_deployment` resource—a `selector` block:

```
  spec {
    replicas = var.replicas

    template {
      metadata {
        labels = local.pod_labels
      }

      spec {
        container {
          name  = var.name
          image = var.image

          port {
            container_port = var.container_port
          }

          dynamic "env" {
            for_each = var.environment_variables
            content {
              name  = env.key
              value = env.value
            }
          }
        }
      }
    }

    selector {
      match_labels = local.pod_labels
    }
  }
```

The `selector` block tells the Kubernetes Deployment what to target. By setting it to `pod_labels`, you are telling it to manage deployments for the Pod Template you just defined. Why doesn't the Deployment just assume that the Pod Template defined within that Deployment is the one you want to target? Well, Kubernetes tries to be an extremely flexible and decoupled system: e.g., it's possible to define a Deployment for Pods that are defined separately, so you always need to specify a `selector` to tell the Deployment what to target.

That wraps up the `kubernetes_deployment` resource. The next step is to use the `kubernetes_service` resource to create a Kubernetes Service:

```
resource "kubernetes_service" "app" {
  metadata {
    name = var.name
  }
```

```
  spec {
    type = "LoadBalancer"
    port {
      port        = 80
      target_port = var.container_port
      protocol    = "TCP"
    }
    selector = local.pod_labels
  }
}
```

Let's go through these parameters:

metadata

Just as with the Deployment object, the Service object uses metadata to identify and target that object in API calls. In the preceding code, I've set the Service name to the `name` input variable.

type

I've configured this Service as type `LoadBalancer`, which, depending on how your Kubernetes cluster is configured, will deploy a different type of load balancer: e.g., in AWS, with EKS, you might get an Elastic Load Balancer, whereas in Google Cloud, with GKE, you might get a Cloud Load Balancer.

port

I'm configuring the load balancer to route traffic on port 80 (the default port for HTTP) to the port the container is listening on.

selector

Just as with the Deployment object, the Service object uses a selector to specify what that Service should be targeting. By setting the selector to `pod_labels`, the Service and the Deployment will both operate on the same Pods.

The final step is to expose the Service endpoint (the load balancer hostname) as an output variable in *outputs.tf*:

```
locals {
  status = kubernetes_service.app.status
}

output "service_endpoint" {
  value = try(
    "http://${local.status[0]["load_balancer"][0]["ingress"][0]["hostname"]}
    "(error parsing hostname from status)"
```

```
    )
    description = "The K8S Service endpoint"
  }
```

This convoluted code needs a bit of explanation. The `kubernetes_service` resource has an output attribute called `status` that returns the latest status of the Service. I've stored this attribute in a local variable called `status`. For a Service of type LoadBalancer, `status` will contain a complicated object that looks something like this:

```
[
  {
    load_balancer = [
      {
        ingress = [
          {
            hostname = "<HOSTNAME>"
          }
        ]
      }
    ]
  }
]
```

Buried within this deeply nested object is the hostname for the load balancer that you want. This is why the `service_endpoint` output variable needs to use a complicated sequence of array lookups (e.g., `[0]`) and map lookups (e.g., `["load_balancer"]`) to extract the hostname. But what happens if the `status` attribute returned by the `kubernetes_service` resource happens to look a little different? In that case, any of those array and map lookups could fail, leading to a confusing error.

To handle this error gracefully, I've wrapped the entire expression in a function called `try`. The `try` function has the following syntax:

```
try(ARG1, ARG2, ..., ARGN)
```

This function evaluates all the arguments you pass to it and returns the first argument that doesn't produce any errors. Therefore, the `service_endpoint` output variable will either end up with a hostname in it (the first argument) or, if reading the hostname caused an error, the variable will instead say "error parsing hostname from status" (the second argument).

OK, that wraps up the `k8s-app` module. To use it, add a new example in *examples/kubernetes-local*, and create a *main.tf* file in it with the following

contents:

```
module "simple_webapp" {
  source = "../../modules/services/k8s-app"

  name           = "simple-webapp"
  image          = "training/webapp"
  replicas       = 2
  container_port = 5000
}
```

This configures the module to deploy the `training/webapp` Docker image you ran earlier, with two replicas listening on port 5000, and to name all the Kubernetes objects (based on their `metadata`) "simple-webapp". To have this module deploy into your local Kubernetes cluster, add the following `provider` block:

```
provider "kubernetes" {
  config_path    = "~/.kube/config"
  config_context = "docker-desktop"
}
```

This code tells the Kubernetes provider to authenticate to your local Kubernetes cluster by using the `docker-desktop` context from your `kubectl` config. Run `terraform apply` to see how it works:

```
$ terraform apply

(...)

Apply complete! Resources: 2 added, 0 changed, 0 destroyed.

Outputs:

service_endpoint = "http://localhost"
```

Give the app a few seconds to boot and then try out that `service_endpoint`:

```
$ curl http://localhost
Hello world!
```

Success!

That said, this looks nearly identical to the output of the `docker run` command, so was all that extra work worth it? Well, let's look under the

hood to see what's going on. You can use `kubectl` to explore your cluster. First, run the `get deployments` command:

```
$ kubectl get deployments
NAME             READY   UP-TO-DATE   AVAILABLE   AGE
simple-webapp    2/2     2            2           3m21s
```

You can see your Kubernetes Deployment, named `simple-webapp`, as that was the name in the `metadata` block. This Deployment is reporting that 2/2 Pods (the two replicas) are ready. To see those Pods, run the `get pods` command:

```
$ kubectl get pods
NAME                              READY   STATUS    RESTARTS   AGE
simple-webapp-d45b496fd-7d447     1/1     Running   0          2m36s
simple-webapp-d45b496fd-vl6j7     1/1     Running   0          2m36s
```

So that's one difference from `docker run` already: there are multiple containers running here, not just one. Moreover, those containers are being actively monitored and managed. For example, if one crashed, a replacement will be deployed automatically. You can see this in action by running the `docker ps` command:

```
$ docker ps
CONTAINER ID   IMAGE              COMMAND           CREATED          STATUS
b60f5147954a   training/webapp    "python app.py"   3 seconds ago    Up 2 sec
c350ec648185   training/webapp    "python app.py"   12 minutes ago   Up 12 mi
```

Grab the `CONTAINER ID` of one of those containers, and use the `docker kill` command to shut it down:

```
$ docker kill b60f5147954a
```

If you run `docker ps` again very quickly, you'll see just one container left running:

```
$ docker ps
CONTAINER ID   IMAGE              COMMAND           CREATED          STATUS
c350ec648185   training/webapp    "python app.py"   12 minutes ago   Up 12 mi
```

But just a few seconds later, the Kubernetes Deployment will have detected that there is only one replica instead of the requested two, and it'll launch a replacement container automatically:

```
$ docker ps
CONTAINER ID   IMAGE              COMMAND           CREATED          STATUS
56a216b8a829   training/webapp    "python app.py"   1 second ago     Up 5 sec
c350ec648185   training/webapp    "python app.py"   12 minutes ago   Up 12 mi
```

So Kubernetes is ensuring that you always have the expected number of replicas running. Moreover, it is also running a load balancer to distribute traffic across those replicas, which you can see by running the `kubectl get services` command:

```
$ kubectl get services
NAME            TYPE           CLUSTER-IP     EXTERNAL-IP   PORT(S)         A
kubernetes      ClusterIP      10.96.0.1      <none>        443/TCP         4
simple-webapp   LoadBalancer   10.110.25.79   localhost     80:30234/TCP    4
```

The first service in the list is Kubernetes itself, which you can ignore. The second is the Service you created, also with the name `simple-webapp` (based on the `metadata` block). This service runs a load balancer for your app: you can see the IP it's accessible at (`localhost`) and the port it's listening on (80).

Kubernetes Deployments also provide automatic rollout of updates. A fun trick with the `training/webapp` Docker image is that if you set the environment variable `PROVIDER` to some value, it'll use that value instead of the word *world* in the text "Hello, world!" Update *examples/kubernetes-local/main.tf* to set this environment variable as follows:

```
module "simple_webapp" {
  source = "../../modules/services/k8s-app"

  name           = "simple-webapp"
  image          = "training/webapp"
  replicas       = 2
  container_port = 5000

  environment_variables = {
    PROVIDER = "Terraform"
  }
}
```

Run `apply` one more time:

```
$ terraform apply

(...)

Apply complete! Resources: 0 added, 1 changed, 0 destroyed.
```

```
Outputs:

service_endpoint = "http://localhost"
```

After a few seconds, try the endpoint again:

```
$ curl http://localhost
Hello Terraform!
```

And there you go, the Deployment has rolled out your change automatically: under the hood, Deployments do a rolling deployment by default, similar to what you saw with Auto Scaling Groups (note that you can change deployment settings by adding a `strategy` block to the `kubernetes_deployment` resource).

Deploying Docker Containers in AWS Using Elastic Kubernetes Service

Kubernetes has one more trick up its sleeve: it's fairly portable. That is, you can reuse both the Docker images and the Kubernetes configurations in a totally different cluster and get similar results. To see this in action, let's now deploy a Kubernetes cluster in AWS.

Setting up and managing a secure, highly available, scalable Kubernetes cluster in the cloud from scratch is complicated. Fortunately, most cloud providers offer managed Kubernetes services, where they run the control plane and worker nodes for you: e.g., Elastic Kubernetes Service (EKS) in AWS, Azure Kubernetes Service (AKS) in Azure, and Google Kubernetes Engine (GKE) in Google Cloud. I'm going to show you how to deploy a very basic EKS cluster in AWS.

Create a new module in *modules/services/eks-cluster*, and define the API for the module in a *variables.tf* file with the following input variables:

```
variable "name" {
  description = "The name to use for the EKS cluster"
  type        = string
}

variable "min_size" {
  description = "Minimum number of nodes to have in the EKS cluster"
  type        = number
}

variable "max_size" {
  description = "Maximum number of nodes to have in the EKS cluster"
```

```
  type        = number
}

variable "desired_size" {
  description = "Desired number of nodes to have in the EKS cluster"
  type        = number
}

variable "instance_types" {
  description = "The types of EC2 instances to run in the node group"
  type        = list(string)
}
```

This code exposes input variables to set the EKS cluster's name, size, and the types of instances to use for the worker nodes. Next, in *main.tf*, create an IAM role for the control plane:

```
# Create an IAM role for the control plane
resource "aws_iam_role" "cluster" {
  name               = "${var.name}-cluster-role"
  assume_role_policy = data.aws_iam_policy_document.cluster_assume_role.json
}

# Allow EKS to assume the IAM role
data "aws_iam_policy_document" "cluster_assume_role" {
  statement {
    effect  = "Allow"
    actions = ["sts:AssumeRole"]
    principals {
      type        = "Service"
      identifiers = ["eks.amazonaws.com"]
    }
  }
}

# Attach the permissions the IAM role needs
resource "aws_iam_role_policy_attachment" "AmazonEKSClusterPolicy" {
  policy_arn = "arn:aws:iam::aws:policy/AmazonEKSClusterPolicy"
  role       = aws_iam_role.cluster.name
}
```

This IAM role can be assumed by the EKS service, and it has a Managed IAM Policy attached that gives the control plane the permissions it needs. Now, add the `aws_vpc` and `aws_subnets` data sources to fetch information about the Default VPC and its subnets:

```
# Since this code is only for learning, use the Default VPC and subnets.
# For real-world use cases, you should use a custom VPC and private subnets.

data "aws_vpc" "default" {
```

```
  default = true
}

data "aws_subnets" "default" {
  filter {
    name   = "vpc-id"
    values = [data.aws_vpc.default.id]
  }
}
```

Now you can create the control plane for the EKS cluster by using the `aws_eks_cluster` resource:

```
resource "aws_eks_cluster" "cluster" {
  name     = var.name
  role_arn = aws_iam_role.cluster.arn
  version  = "1.21"

  vpc_config {
    subnet_ids = data.aws_subnets.default.ids
  }

  # Ensure that IAM Role permissions are created before and deleted after
  # the EKS Cluster. Otherwise, EKS will not be able to properly delete
  # EKS managed EC2 infrastructure such as Security Groups.
  depends_on = [
    aws_iam_role_policy_attachment.AmazonEKSClusterPolicy
  ]
}
```

The preceding code configures the control plane to use the IAM role you just created, and to deploy into the Default VPC and subnets.

Next up are the worker nodes. EKS supports several different types of worker nodes: self-managed EC2 Instances (e.g., in an ASG that you create), AWS-managed EC2 Instances (known as a *managed node group*), and Fargate (serverless).[3] The simplest option to use for the examples in this chapter will be the managed node groups.

To deploy a managed node group, you first need to create another IAM role:

```
# Create an IAM role for the node group
resource "aws_iam_role" "node_group" {
  name               = "${var.name}-node-group"
  assume_role_policy = data.aws_iam_policy_document.node_assume_role.json
}

# Allow EC2 instances to assume the IAM role
```

```
data "aws_iam_policy_document" "node_assume_role" {
  statement {
    effect  = "Allow"
    actions = ["sts:AssumeRole"]
    principals {
      type        = "Service"
      identifiers = ["ec2.amazonaws.com"]
    }
  }
}

# Attach the permissions the node group needs
resource "aws_iam_role_policy_attachment" "AmazonEKSWorkerNodePolicy" {
  policy_arn = "arn:aws:iam::aws:policy/AmazonEKSWorkerNodePolicy"
  role       = aws_iam_role.node_group.name
}

resource "aws_iam_role_policy_attachment" "AmazonEC2ContainerRegistryReadOnl
  policy_arn = "arn:aws:iam::aws:policy/AmazonEC2ContainerRegistryReadOnly"
  role       = aws_iam_role.node_group.name
}

resource "aws_iam_role_policy_attachment" "AmazonEKS_CNI_Policy" {
  policy_arn = "arn:aws:iam::aws:policy/AmazonEKS_CNI_Policy"
  role       = aws_iam_role.node_group.name
}
```

This IAM role can be assumed by the EC2 service (which makes sense, as managed node groups use EC2 Instances under the hood), and it has several Managed IAM Policies attached that give the managed node group the permissions it needs. Now you can use the `aws_eks_node_group` resource to create the managed node group itself:

```
resource "aws_eks_node_group" "nodes" {
  cluster_name    = aws_eks_cluster.cluster.name
  node_group_name = var.name
  node_role_arn   = aws_iam_role.node_group.arn
  subnet_ids      = data.aws_subnets.default.ids
  instance_types  = var.instance_types

  scaling_config {
    min_size     = var.min_size
    max_size     = var.max_size
    desired_size = var.desired_size
  }

  # Ensure that IAM Role permissions are created before and deleted after
  # the EKS Node Group. Otherwise, EKS will not be able to properly
  # delete EC2 Instances and Elastic Network Interfaces.
  depends_on = [
    aws_iam_role_policy_attachment.AmazonEKSWorkerNodePolicy,
```

```
    aws_iam_role_policy_attachment.AmazonEC2ContainerRegistryReadOnly,
    aws_iam_role_policy_attachment.AmazonEKS_CNI_Policy,
  ]
}
```

This code configures the managed node group to use the control plane and IAM role you just created, to deploy into the Default VPC, and to use the name, size, and instance type parameters passed in as input variables.

In *outputs.tf*, add the following output variables:

```
output "cluster_name" {
  value       = aws_eks_cluster.cluster.name
  description = "Name of the EKS cluster"
}

output "cluster_arn" {
  value       = aws_eks_cluster.cluster.arn
  description = "ARN of the EKS cluster"
}

output "cluster_endpoint" {
  value       = aws_eks_cluster.cluster.endpoint
  description = "Endpoint of the EKS cluster"
}

output "cluster_certificate_authority" {
  value       = aws_eks_cluster.cluster.certificate_authority
  description = "Certificate authority of the EKS cluster"
}
```

OK, the `eks-cluster` module is now ready to roll. Let's use it and the `k8s-app` module from earlier to deploy an EKS cluster and to deploy the `training/webapp` Docker image into that cluster. Create *examples/kubernetes-eks/main.tf*, and configure the `eks-cluster` module as follows:

```
provider "aws" {
  region = "us-east-2"
}

module "eks_cluster" {
  source = "../../modules/services/eks-cluster"

  name         = "example-eks-cluster"
  min_size     = 1
  max_size     = 2
  desired_size = 1

  # Due to the way EKS works with ENIs, t3.small is the smallest
  # instance type that can be used for worker nodes. If you try
```

```
    # something smaller like t2.micro, which only has 4 ENIs,
    # they'll all be used up by system services (e.g., kube-proxy)
    # and you won't be able to deploy your own Pods.
    instance_types = ["t3.small"]
  }
```

Next, configure the `k8s-app` module as follows:

```
  provider "kubernetes" {
    host = module.eks_cluster.cluster_endpoint
    cluster_ca_certificate = base64decode(
      module.eks_cluster.cluster_certificate_authority[0].data
    )
    token = data.aws_eks_cluster_auth.cluster.token
  }

  data "aws_eks_cluster_auth" "cluster" {
    name = module.eks_cluster.cluster_name
  }

  module "simple_webapp" {
    source = "../../modules/services/k8s-app"

    name           = "simple-webapp"
    image          = "training/webapp"
    replicas       = 2
    container_port = 5000

    environment_variables = {
      PROVIDER = "Terraform"
    }

    # Only deploy the app after the cluster has been deployed
    depends_on = [module.eks_cluster]
  }
```

The preceding code configures the Kubernetes provider to authenticate to the EKS cluster, rather than your local Kubernetes cluster (from Docker Desktop). It then uses the `k8s-app` module to deploy the `training/webapp` Docker image exactly the same way as you did when deploying it to Docker Desktop; the only difference is the addition of the `depends_on` parameter to ensure that Terraform only tries to deploy the Docker image after the EKS cluster has been deployed.

Next, pass through the service endpoint as an output variable:

```
  output "service_endpoint" {
    value       = module.simple_webapp.service_endpoint
```

```
  description = "The K8S Service endpoint"
}
```

OK, now you're ready to deploy! Run `terraform apply` as usual (note that EKS clusters can take 10–20 minutes to deploy, so be patient):

```
$ terraform apply

(...)

Apply complete! Resources: 10 added, 0 changed, 0 destroyed.

Outputs:

service_endpoint = "http://774696355.us-east-2.elb.amazonaws.com"
```

Wait a little while for the web app to spin up and pass health checks, and then test out the `service_endpoint`:

```
$ curl http://774696355.us-east-2.elb.amazonaws.com
Hello Terraform!
```

And there you have it! The same Docker image and Kubernetes code is now running in an EKS cluster in AWS, just the way it ran on your local computer. All the same features work here too. For example, try updating `environment_variables` to a different `PROVIDER` value, such as "Readers":

```
module "simple_webapp" {
  source = "../../modules/services/k8s-app"

  name           = "simple-webapp"
  image          = "training/webapp"
  replicas       = 2
  container_port = 5000

  environment_variables = {
    PROVIDER = "Readers"
  }

  # Only deploy the app after the cluster has been deployed
  depends_on = [module.eks_cluster]
}
```

Rerun `apply`, and just a few seconds later, the Kubernetes Deployment will have deployed the changes:

```
$ curl http://774696355.us-east-2.elb.amazonaws.com
Hello Readers!
```

This is one of the advantages of using Docker: changes can be deployed very quickly.

You can use `kubectl` again to see what's happening in your cluster. To authenticate `kubectl` to the EKS cluster, you can use the `aws eks update-kubeconfig` command to automatically update your *$HOME/.kube/config* file:

```
$ aws eks update-kubeconfig --region <REGION> --name <EKS_CLUSTER_NAME>
```

where `REGION` is the AWS region and `EKS_CLUSTER_NAME` is the name of your EKS cluster. In the Terraform module, you deployed to the `us-east-2` region and named the cluster `kubernetes-example`, so the command will look like this:

```
$ aws eks update-kubeconfig --region us-east-2 --name kubernetes-example
```

Now, just as before, you can use the `get nodes` command to inspect the worker nodes in your cluster, but this time, add the `-o wide` flag to get a bit more info:

```
$ kubectl get nodes
NAME                                STATUS   AGE   EXTERNAL-IP     OS-IMAGE
xxx.us-east-2.compute.internal   Ready    22m   3.134.78.187   Amazon Linux
```

The preceding snippet is highly truncated to fit into the book, but in the real output, you should be able to see the one worker node, its internal and external IP, version information, OS information, and much more.

You can use the `get deployments` command to inspect your Deployments:

```
$ kubectl get deployments
NAME            READY   UP-TO-DATE   AVAILABLE   AGE
simple-webapp   2/2     2            2           19m
```

Next, run `get pods` to see the Pods:

```
$ kubectl get pods
NAME            READY   UP-TO-DATE   AVAILABLE   AGE
simple-webapp   2/2     2            2           19m
```

And finally, run `get services` to see the Services:

```
$ kubectl get services
NAME            TYPE           EXTERNAL-IP                                   PORT
kubernetes      ClusterIP      <none>                                        443/
simple-webapp   LoadBalancer   774696355.us-east-2.elb.amazonaws.com         80/T
```

You should be able to see your load balancer and the URL you used to test it.

So there you have it: two different providers, both working in the same cloud, helping you to deploy containerized workloads.

That said, just as in previous sections, I want to leave you with a few warnings:

Warning 1: These Kubernetes examples are very simplified!

Kubernetes is complicated, and it's rapidly evolving and changing; trying to explain all the details can easily fill a book all by itself. Since this is a book about Terraform, and not Kubernetes, my goal with the Kubernetes examples in this chapter was to keep them as simple and minimal as possible. Therefore, while I hope the code examples you've seen have been useful from a learning and experimentation perspective, if you are going to use Kubernetes for real-world, production use cases, you'll need to change many aspects of this code, such as configuring a number of additional services and settings in the `eks-cluster` module (e.g., ingress controllers, secret envelope encryption, security groups, OIDC authentication, Role-Based Access Control (RBAC) mapping, VPC CNI, kube-proxy, CoreDNS), exposing many other settings in the `k8s-app` module (e.g., secrets management, volumes, liveness probes, readiness probes, labels, annotations, multiple ports, multiple containers), and using a custom VPC with private subnets for your EKS cluster instead of the Default VPC and public subnets.[4]

Warning 2: Use multiple providers sparingly

Although you certainly can use multiple providers in a single module, I don't recommend doing it too often, for similar reasons to why I don't recommend using provider aliases too often: in most cases, you want each provider to be isolated in its own module so that you can manage it separately and limit the blast radius from mistakes or attackers.

Moreover, Terraform doesn't have great support for dependency ordering between providers. For example, in the Kubernetes exam-

ple, you had a single module that deployed both the EKS cluster, using the AWS Provider, and a Kubernetes app into that cluster, using the Kubernetes provider. As it turns out, the Kubernetes provider documentation explicitly recommends *against* this pattern:

> *When using interpolation to pass credentials to the Kubernetes provider from other resources, these resources SHOULD NOT be created in the same Terraform module where Kubernetes provider resources are also used. This will lead to intermittent and unpredictable errors which are hard to debug and diagnose. The root issue lies with the order in which Terraform itself evaluates the provider blocks vs. actual resources.*

The example code in this book is able to work around these issues by depending on the `aws_eks_cluster_auth` data source, but that's a bit of a hack. Therefore, in production code, I always recommend deploying the EKS cluster in one module and deploying Kubernetes apps in separate modules, after the cluster has been deployed.

Conclusion

At this point, you hopefully understand how to work with multiple providers in Terraform code, and you can answer the three questions from the beginning of this chapter:

What if you need to deploy to multiple AWS regions?

Use multiple `provider` blocks, each configured with a different `region` and `alias` parameter.

What if you need to deploy to multiple AWS accounts?

Use multiple `provider` blocks, each configured with a different `assume_role` block and an `alias` parameter.

What if you need to deploy to other clouds, such as Azure or GCP or Kubernetes?

Use multiple `provider` blocks, each configured for its respective cloud.

However, you've also seen that using multiple providers in one module is typically an antipattern. So the real answer to these questions, especially in real-world, production use cases, is to use each provider in a separate

module to keep different regions, accounts, and clouds isolated from one another, and to limit your blast radius.

Let's now move on to Chapter 8, where I'll go over several other patterns for how to build Terraform modules for real-world, production use cases—the kind of modules you could bet your company on.

1 In fact, you could even skip the `provider` block and just add any resource or data source from an official provider and Terraform will figure out which provider to use based on the prefix: for example, if you add the `aws_instance` resource, Terraform will know to use the AWS Provider based on the `aws_` prefix.

2 See "Multi-Cloud is the Worst Practice".

3 For a comparison of the different types of EKS worker nodes, see the Gruntwork blog.

4 Alternatively, you can use off-the-shelf production-grade Kubernetes modules, such as the ones in the Gruntwork Infrastructure as Code Library.