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

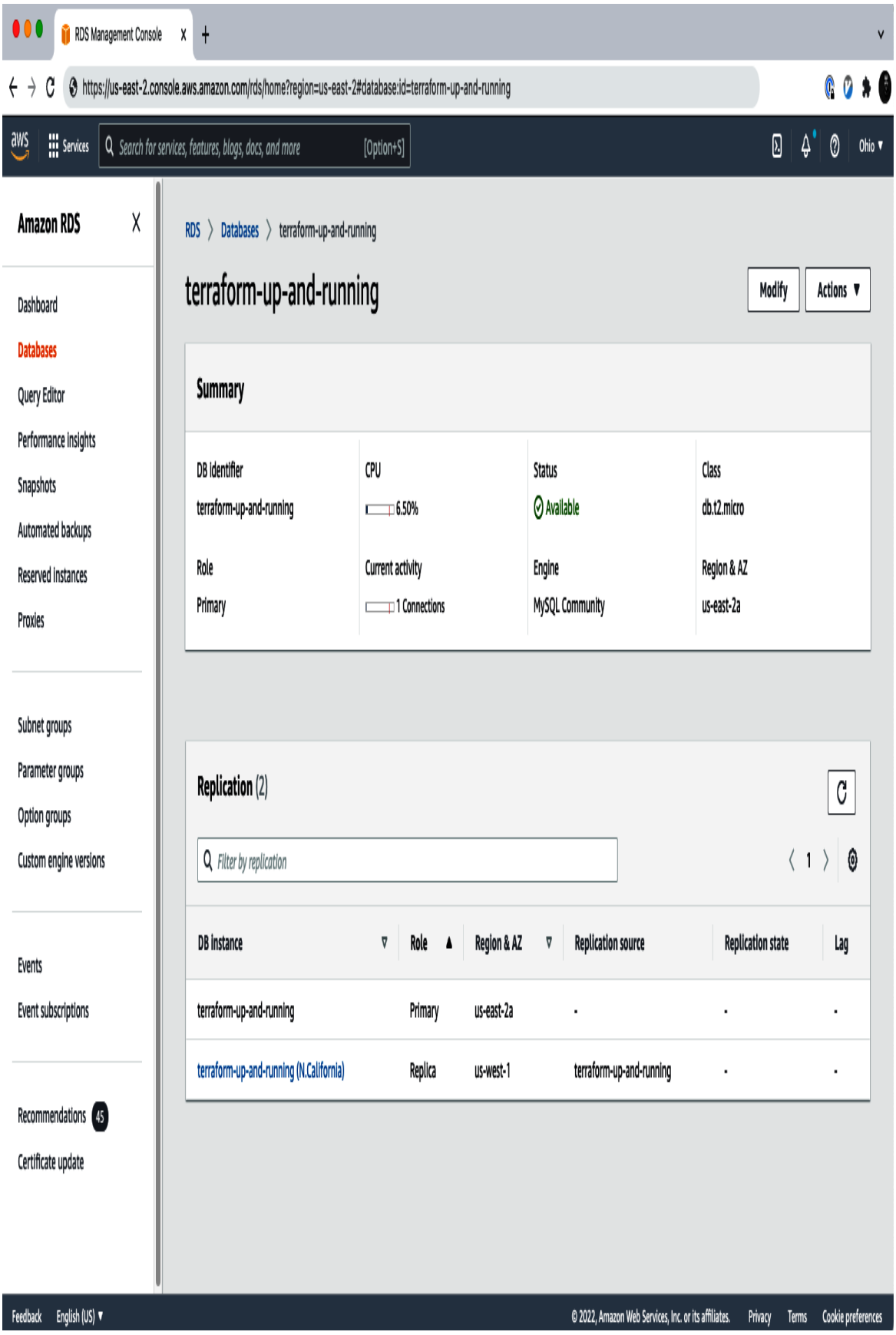
RDS Management Console
https://us-east-2.console.aws.amazon.com/rds/home?region=us-east-2#database:id=terraform-up-and-running
Services
Search for services, features, blogs, docs, and more [Option+S]
Ohio
Amazon RDS
Dashboard
Databases
Query Editor
Performance Insights
Snapshots
Automated backups
Reserved instances
Proxies
Subnet groups
Parameter groups
Option groups
Custom engine versions
Events
Event subscriptions
Recommendations 45
Certificate update
RDS > Databases > terraform-up-and-running
terraform-up-and-running
Modify
Actions
Summary
DB identifier
terraform-up-and-running
CPU
6.50%
Status
Available
Class
db.t2.micro
Role
Primary
Current activity
1 Connections
Engine
MySQL Community
Region & AZ
us-east-2a
Replication (2)
Filter by replication
1
DB instance
Role
Region & AZ
Replication source
Replication state
Lag
terraform-up-and-running
Primary
us-east-2a
terraform-up-and-running (N.California)
Replica
us-west-1
terraform-up-and-running
Feedback
English (US)
© 2022, Amazon Web Services, Inc. or its affiliates.
Privacy
Terms
Cookie preferences

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 multiregion 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.