As soon as data is shared, you run into a new problem: locking. Without locking, if two team members are running Terraform at the same time, you can run into race conditions as multiple Terraform processes make concurrent updates to the state files, leading to conflicts, data loss, and state file corruption.

Isolating state files

When making changes to your infrastructure, it's a best practice to isolate different environments. For example, when making a change in a testing or staging environment, you want to be sure that there is no way you can accidentally break production. But how can you isolate your changes if all of your infrastructure is defined in the same Terraform state file?

In the following sections, I'll dive into each of these problems and show you how to solve them.

Shared Storage for State Files

The most common technique for allowing multiple team members to access a common set of files is to put them in version control (e.g., Git). Although you should definitely store your Terraform code in version control, storing Terraform state in version control is a *bad idea* for the following reasons:

Manual error

It's too easy to forget to pull down the latest changes from version control before running Terraform or to push your latest changes to version control after running Terraform. It's just a matter of time before someone on your team runs Terraform with out-of-date state files and, as a result, accidentally rolls back or duplicates previous deployments.

Locking

Most version control systems do not provide any form of locking that would prevent two team members from running `terraform apply` on the same state file at the same time.

Secrets

All data in Terraform state files is stored in plain text. This is a problem because certain Terraform resources need to store sensitive data. For example, if you use the `aws_db_instance` resource to create a database, Terraform will store the username and password for the database in a state file in plain text, and you shouldn't store plain text secrets in version control.

Instead of using version control, the best way to manage shared storage for state files is to use Terraform's built-in support for remote backends. A Terraform *backend* determines how Terraform loads and stores state. The default backend, which you've been using this entire time, is the *local backend*, which stores the state file on your local disk. *Remote backends* allow you to store the state file in a remote, shared store. A number of remote backends are supported, including Amazon S3, Azure Storage, Google Cloud Storage, and HashiCorp's Terraform Cloud and Terraform Enterprise.

Remote backends solve the three issues just listed:

Manual error

After you configure a remote backend, Terraform will automatically load the state file from that backend every time you run `plan` or `apply`, and it'll automatically store the state file in that backend after each `apply`, so there's no chance of manual error.

Locking

Most of the remote backends natively support locking. To run `terraform apply`, Terraform will automatically acquire a lock; if someone else is already running `apply`, they will already have the

lock, and you will have to wait. You can run `apply` with the `-lock-timeout=<TIME>` parameter to instruct Terraform to wait up to `TIME` for a lock to be released (e.g., `-lock-timeout=10m` will wait for 10 minutes).

Secrets

Most of the remote backends natively support encryption in transit and encryption at rest of the state file. Moreover, those backends usually expose ways to configure access permissions (e.g., using IAM policies with an Amazon S3 bucket), so you can control who has access to your state files and the secrets they might contain. It would be better still if Terraform natively supported encrypting secrets within the state file, but these remote backends reduce most of the security concerns, given that at least the state file isn't stored in plain text on disk anywhere.

If you're using Terraform with AWS, Amazon S3 (Simple Storage Service), which is Amazon's managed file store, is typically your best bet as a remote backend for the following reasons:

- It's a managed service, so you don't need to deploy and manage extra infrastructure to use it.
- It's designed for 99.999999999% durability and 99.99% availability, which means you don't need to worry too much about data loss or outages.[1]
- It supports encryption, which reduces worries about storing sensitive data in state files. You still have to be very careful who on your team can access the S3 bucket, but at least the data will be encrypted at rest (Amazon S3 supports server-side encryption using AES-256) and in transit (Terraform uses TLS when talking to Amazon S3).
- It supports locking via DynamoDB. (More on this later.)
- It supports *versioning*, so every revision of your state file is stored, and you can roll back to an older version if something goes wrong.

- It's inexpensive, with most Terraform usage easily fitting into the AWS Free Tier.[2]

To enable remote state storage with Amazon S3, the first step is to create an S3 bucket. Create a *main.tf* file in a new folder (it should be a different folder from where you store the configurations from Chapter 2), and at the top of the file, specify AWS as the provider:

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

Next, create an S3 bucket by using the `aws_s3_bucket` resource:

```
resource "aws_s3_bucket" "terraform_state" {
  bucket = "terraform-up-and-running-state"

  # Prevent accidental deletion of this S3 bucket
  lifecycle {
    prevent_destroy = true
  }
}
```

This code sets the following arguments:

`bucket`

> This is the name of the S3 bucket. Note that S3 bucket names must be *globally* unique among all AWS customers. Therefore, you will need to change the `bucket` parameter from `"terraform-up-and-running-state"` (which I already created) to your own name. Make sure to remember this name and take note of what AWS region you're using because you'll need both pieces of information again a little later on.

`prevent_destroy`

> `prevent_destroy` is the second lifecycle setting you've seen (the first was `create_before_destroy` in Chapter 2). When you set

`prevent_destroy` to `true` on a resource, any attempt to delete that resource (e.g., by running `terraform destroy`) will cause Terraform to exit with an error. This is a good way to prevent accidental deletion of an important resource, such as this S3 bucket, which will store all of your Terraform state. Of course, if you really mean to delete it, you can just comment that setting out.

Let's now add several extra layers of protection to this S3 bucket.

First, use the `aws_s3_bucket_versioning` resource to enable versioning on the S3 bucket so that every update to a file in the bucket actually creates a new version of that file. This allows you to see older versions of the file and revert to those older versions at any time, which can be a useful fallback mechanism if something goes wrong:

```
# Enable versioning so you can see the full revision history of
your
# state files
resource "aws_s3_bucket_versioning" "enabled" {
  bucket = aws_s3_bucket.terraform_state.id
  versioning_configuration {
    status = "Enabled"
  }
}
```

Second, use the `aws_s3_bucket_server_side_encryption_configuration` resource to turn server-side encryption on by default for all data written to this S3 bucket. This ensures that your state files, and any secrets they might contain, are always encrypted on disk when stored in S3:

```
# Enable server-side encryption by default
resource "aws_s3_bucket_server_side_encryption_configuration"
"default" {
  bucket = aws_s3_bucket.terraform_state.id

  rule {
    apply_server_side_encryption_by_default {
      sse_algorithm = "AES256"
```

```
    }
  }
}
```

Third, use the `aws_s3_bucket_public_access_block` resource to block all public access to the S3 bucket. S3 buckets are private by default, but as they are often used to serve static content—e.g., images, fonts, CSS, JS, HTML—it is possible, even easy, to make the buckets public. Since your Terraform state files may contain sensitive data and secrets, it's worth adding this extra layer of protection to ensure no one on your team can ever accidentally make this S3 bucket public:

```
# Explicitly block all public access to the S3 bucket
resource "aws_s3_bucket_public_access_block" "public_access" {
  bucket                  = aws_s3_bucket.terraform_state.id
  block_public_acls       = true
  block_public_policy     = true
  ignore_public_acls      = true
  restrict_public_buckets = true
}
```

Next, you need to create a DynamoDB table to use for locking. DynamoDB is Amazon's distributed key-value store. It supports strongly consistent reads and conditional writes, which are all the ingredients you need for a distributed lock system. Moreover, it's completely managed, so you don't have any infrastructure to run yourself, and it's inexpensive, with most Terraform usage easily fitting into the AWS Free Tier.[3]

To use DynamoDB for locking with Terraform, you must create a DynamoDB table that has a primary key called `LockID` (with this *exact* spelling and capitalization). You can create such a table using the `aws_dynamodb_table` resource:

```
resource "aws_dynamodb_table" "terraform_locks" {
  name         = "terraform-up-and-running-locks"
  billing_mode = "PAY_PER_REQUEST"
  hash_key     = "LockID"

  attribute {
    name = "LockID"
```

```
    type = "S"
  }
}
```

Run `terraform init` to download the provider code, and then run `terraform apply` to deploy. After everything is deployed, you will have an S3 bucket and DynamoDB table, but your Terraform state will still be stored locally. To configure Terraform to store the state in your S3 bucket (with encryption and locking), you need to add a `backend` configuration to your Terraform code. This is configuration for Terraform itself, so it resides within a `terraform` block and has the following syntax:

```
terraform {
  backend "<BACKEND_NAME>" {
    [CONFIG...]
  }
}
```

where `BACKEND_NAME` is the name of the backend you want to use (e.g., `"s3"`) and `CONFIG` consists of one or more arguments that are specific to that backend (e.g., the name of the S3 bucket to use). Here's what the `backend` configuration looks like for an S3 bucket:

```
terraform {
  backend "s3" {
    # Replace this with your bucket name!
    bucket         = "terraform-up-and-running-state"
    key            = "global/s3/terraform.tfstate"
    region         = "us-east-2"

    # Replace this with your DynamoDB table name!
    dynamodb_table = "terraform-up-and-running-locks"
    encrypt        = true
  }
}
```

Let's go through these settings one at a time:

`bucket`

The name of the S3 bucket to use. Make sure to replace this with the name of the S3 bucket you created earlier.

`key`

The filepath within the S3 bucket where the Terraform state file should be written. You'll see a little later on why the preceding example code sets this to `global/s3/terraform.tfstate`.

`region`

The AWS region where the S3 bucket lives. Make sure to replace this with the region of the S3 bucket you created earlier.

`dynamodb_table`

The DynamoDB table to use for locking. Make sure to replace this with the name of the DynamoDB table you created earlier.

`encrypt`

Setting this to `true` ensures that your Terraform state will be encrypted on disk when stored in S3. We already enabled default encryption in the S3 bucket itself, so this is here as a second layer to ensure that the data is always encrypted.

To instruct Terraform to store your state file in this S3 bucket, you're going to use the `terraform init` command again. This command not only can download provider code, but also configure your Terraform backend (and you'll see yet another use later on, too). Moreover, the `init` command is idempotent, so it's safe to run it multiple times:

```
$ terraform init

Initializing the backend...
Acquiring state lock. This may take a few moments...
Do you want to copy existing state to the new backend?
  Pre-existing state was found while migrating the previous
"local" backend
```

```
  to the newly configured "s3" backend. No existing state was
found in the
  newly configured "s3" backend. Do you want to copy this state
to the new
  "s3" backend? Enter "yes" to copy and "no" to start with an
empty state.

  Enter a value:
```

Terraform will automatically detect that you already have a state file locally and prompt you to copy it to the new S3 backend. If you type **`yes`**, you should see the following:

```
Successfully configured the backend "s3"! Terraform will
automatically
use this backend unless the backend configuration changes.
```

After running this command, your Terraform state will be stored in the S3 bucket. You can check this by heading over to the S3 Management Console in your browser and clicking your bucket. You should see something similar to Figure 3-1.

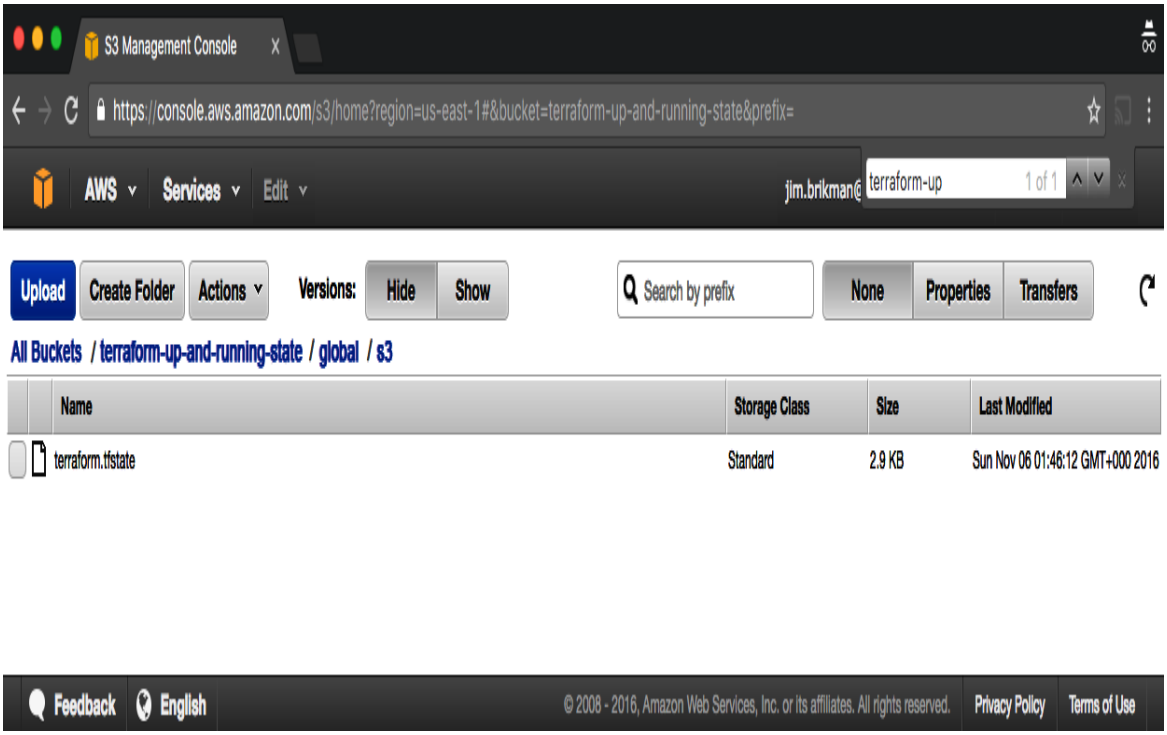

Figure 3-1. You can use the AWS Console to see how your state file is stored in an S3 bucket.

With this backend enabled, Terraform will automatically pull the latest state from this S3 bucket before running a command and automatically push the latest state to the S3 bucket after running a command. To see this in action, add the following output variables:

```
output "s3_bucket_arn" {
  value       = aws_s3_bucket.terraform_state.arn
  description = "The ARN of the S3 bucket"
}

output "dynamodb_table_name" {
  value       = aws_dynamodb_table.terraform_locks.name
  description = "The name of the DynamoDB table"
}
```

These variables will print out the Amazon Resource Name (ARN) of your S3 bucket and the name of your DynamoDB table. Run `terraform apply` to see it:

```
$ terraform apply

(...)

Acquiring state lock. This may take a few moments...

aws_dynamodb_table.terraform_locks: Refreshing state...
aws_s3_bucket.terraform_state: Refreshing state...

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

Releasing state lock. This may take a few moments...

Outputs:

dynamodb_table_name = "terraform-up-and-running-locks"
s3_bucket_arn = "arn:aws:s3:::terraform-up-and-running-state"
```

Note how Terraform is now acquiring a lock before running `apply` and releasing the lock after!

Now, head over to the S3 console again, refresh the page, and click the gray Show button next to Versions. You should now see several versions of your