





Terraform: Up and Running

PREV
2. Getting Started with Terraform



NEXT
State Reusable Infrastructure with Terraform Modules

Chapter 3. How to Manage Terraform State

In [Chapter 2](#), as you were using Terraform to create and update resources, you may have noticed that every time you ran `terraform plan` or `terraform apply`, Terraform was able to find the resources it created previously and update them accordingly. But how did Terraform know which resources it was supposed to manage? You could have all sorts of infrastructure in your AWS account, deployed through a variety of mechanisms (some manually, some via Terraform, some via the CLI), so how does Terraform know which infrastructure it's responsible for?

In this chapter, you're going to see how Terraform tracks the state of your infrastructure and the impact that has on file layout, isolation, and locking in a Terraform project. Here are the key topics I'll go over:

- What is Terraform state?
- Shared storage for state files
- Locking state files
- Isolating state files
- File layout
- Read-only state

EXAMPLE CODE

As a reminder, all of the code examples in the book can be found at the following URL:
<https://github.com/brikis98/terraform-up-and-running-code>.

What Is Terraform State?

Every time you run Terraform, it records information about what infrastructure it created in a *Terraform state file*. By default, when you run Terraform in the folder */foo/bar*, Terraform creates the file */foo/bar/terraform.tfstate*. This file contains a custom JSON format that records a mapping from the Terraform resources in your configuration files to the representation of those resources in the real world. For example, let's say your Terraform configuration contained the following:

```
resource "aws_instance" "example" {
  ami           = "ami-4bd28157"
  instance_type = "t2.micro"
}
```

After running `terraform apply`, here is a small snippet of the contents of the *terraform.tfstate* file:

```
{
  "aws_instance.example": {
    "type": "aws_instance",
    "primary": {
      "id": "i-66ba8957",
      "attributes": [
```

Find answers on the fly, or master something new. Subscribe today. [See pricing options.](#)

```

    "instance_type": "t2.micro",
    "network_interface_id": "eni-7c4fcf6e",
    "private_dns": "ip-172-31-53-99.ec2.internal",
    "private_ip": "172.31.53.99",
    "public_dns": "ec2-54-159-88-79.compute-1.amazonaws.com",
    "public_ip": "54.159.88.79",
    "subnet_id": "subnet-3b29db10"
  }
}
}

```

Using this simple JSON format, Terraform knows that `aws_instance.example` corresponds to an EC2 Instance in your AWS account with ID `i-66ba8957`. Every time you run Terraform, it can fetch the latest status of this EC2 Instance from AWS and compare that to what's in your Terraform configurations to determine what changes need to be applied.

THE STATE FILE IS A PRIVATE API

The state file format is a private API that changes with every release and is meant only for internal use within Terraform. You should never edit the Terraform state files by hand or write code that reads them directly.

If for some reason you need to manipulate the state file—which should be a relatively rare occurrence—use the `terraform import` command (you'll see an example of this in [Chapter 5](#)) or the `terraform state` command (this is only for advanced use cases).

If you're using Terraform for a personal project, storing state in a local `terraform.tfstate` file works just fine. But if you want to use Terraform as a team on a real product, you run into several problems:

Shared storage for state files

To be able to use Terraform to update your infrastructure, each of your team members needs access to the same Terraform state files. That means you need to store those files in a shared location.

Locking state files

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 may 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). With Terraform state, this is a *bad idea* for two 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.

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. Storing plain-text secrets *anywhere* is a bad idea, including version control. As of November, 2016, this is an [open issue](#) in the Terraform community, although there are some reasonable workarounds, as I will discuss shortly.

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 State Storage*. Using the `terraform remote config` command, you can configure Terraform to fetch and store state data from a remote store every time it runs. Several remote stores are supported, such as Amazon S3, Azure Storage, HashiCorp Consul, and HashiCorp's Terraform Pro and Terraform Enterprise.

I typically recommend Amazon S3 (Simple Storage Service), which is Amazon's managed file store, for the following reasons:

- It's a managed service, so you don't have to deploy and manage extra infrastructure to use it.
- It's designed for 99.999999999% durability and 99.99% availability, which effectively means it'll never lose your data or go down. ¹
- It supports encryption, which reduces worries about storing sensitive data in state files. Anyone on your team who has access to that S3 bucket will be able to see the state files in an unencrypted form, so this is still a partial solution, but at least the data will be encrypted at rest (S3 supports server-side encryption using AES-256) and in transit (Terraform uses SSL to read and write data in S3).
- It supports *versioning*, so every revision of your state file is stored, and you can always roll back to an older version if something goes wrong.
- It's inexpensive, with most Terraform usage easily fitting into the free tier. ²

S3 AND LARGE, DISTRIBUTED TEAMS

S3 is an eventually consistent file store, which means changes can take a few seconds to propagate. If you have a large, geographically distributed team that makes frequent changes to the same Terraform state, there is a very small chance you will end up with stale state. For these sorts of use cases, you may want to use an alternate remote state store, such as Terraform Pro or Terraform Enterprise.

To enable remote state storage with 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-1"
}
```

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

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

  versioning {
    enabled = true
  }

  lifecycle {
    prevent_destroy = true
  }
}
```

This code sets three parameters:

bucket

This is the name of the S3 bucket. Note that it must be *globally* unique. Therefore, you will have to change the `bucket` parameter from `"terraform-up-and-running-state"` (which I already created) to your own name. ³ Make sure to remember this name and take note of what AWS region you're using, as you'll need both pieces of information again a little later on.

versioning

This block enables 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.

prevent_destroy

`prevent_destroy` is the second `lifecycle` setting you've seen (the first was `create_before_destroy`). 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.

Run `terraform plan`, and if everything looks OK, create the bucket by running `terraform apply`. After this completes, you will have an S3 bucket, but your Terraform state is still stored locally. To configure Terraform to store the state in your S3 bucket (with encryption), run the following command, filling in your own values where specified:

```
> terraform remote config \
  -backend=s3 \
  -backend-config="bucket=(YOUR_BUCKET_NAME)" \
  -backend-config="key=global/s3/terraform.tfstate" \
  -backend-config="region=us-east-1" \
  -backend-config="encrypt=true"

Remote configuration updated
Remote state configured and pulled.
```

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

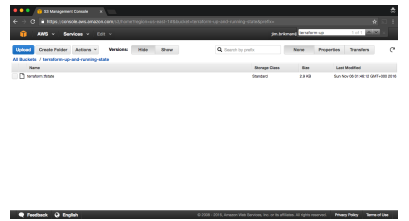


Figure 3-1. Terraform state file stored in S3

With remote state 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 variable:

```
output "s3_bucket_arn" {
  value = "${aws_s3_bucket.terraform_state.arn}"
}
```

This variable will print out the Amazon Resource Name (ARN) of your S3 bucket. Run `terraform apply` to see it:

```
> terraform apply

aws_s3_bucket.terraform_state: Refreshing state...
(ID: terraform-up-and-running-state)

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

Outputs:

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

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 `terraform.tfstate` file in the S3 bucket, as shown in Figure 3-2.

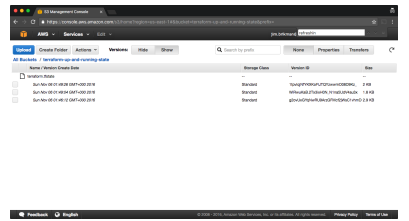


Figure 3-2. Multiple versions of the Terraform state file in S3

This means that Terraform is automatically pushing and pulling state data to and from S3 and S3 is storing every revision of the state file, which can be useful for debugging and rolling back to older versions if something goes wrong.

Locking State Files

Enabling remote state solves the problem of how you share state files with your teammates, but it creates two new problems:

- 1. Each developer on your team needs to remember to run the `terraform remote config` command for every Terraform project. It's easy to mess up or forget to

run this long command.

2. While Terraform remote state storage ensures your state is stored in a shared location, it does *not* provide locking for that shared location. Therefore, race conditions are still possible if two developers are using Terraform at the same time on the same state files.

Future versions of Terraform may solve both of these problems by introducing the concept of “backends,” but in the meantime, you can pick from one of these other solutions:

Terraform Pro or Terraform Enterprise

While Terraform itself is open source, HashiCorp, the company that created Terraform, offers paid options called Terraform Pro and Terraform Enterprise, each of which supports locking for state files.

Build server

To remove the need for locking entirely, you can enforce a rule in your team that no one can run Terraform locally to modify a shared environment (e.g., staging, production). Instead, all the changes must be applied automatically by a *build server*, such as Jenkins or CircleCI, which you can configure to never apply more than one change concurrently. Using a build server to automate deployments is a good idea regardless of the locking strategy you use, as it allows you to catch bugs and enforce compliance rules by running automated tests before applying any change. I’ll come back to build servers in Chapter 6.

Terragrunt

Terragrunt is a thin, open source wrapper for Terraform that configures remote state automatically and provides locking by using Amazon DynamoDB. DynamoDB is part of the AWS free tier, so using it for locking should be free for most teams.

The easiest solution to start with is Terragrunt, since it’s free and does not require setting up any extra infrastructure. To try it out, head over to the Terragrunt GitHub page and follow the instructions in the Readme to install the appropriate Terragrunt binary for your operating system. Next, create a file called *.terragrunt* in the same folder as the Terraform configuration for your S3 bucket, and put the following code in it, filling in your own values where specified:

```
# Configure Terragrunt to use DynamoDB for Locking
lock = {
  backend = "dynamodb"

  config {
    state_file_id = "global/s3"
  }
}

# Configure Terragrunt to automatically store tfstate files in S3
remote_state = {
  backend = "s3"

  config {
    encrypt = "true"
    bucket = "(YOUR_BUCKET_NAME)"
    key    = "global/s3/terraform.tfstate"
    region = "us-east-1"
  }
}
```

The *.terragrunt* file uses the same language as Terraform, HCL. The first part of the configuration tells Terragrunt to use DynamoDB for locking. The *state_file_id* should be unique for each set of Terraform configurations, so they each have their own lock. The second part of the configuration tells Terragrunt to use an S3 bucket for remote state storage using the exact same settings as the `terraform remote config` command you ran earlier.

Once you check this *.terragrunt* file into source control, everyone on your team can use Terragrunt to run all the standard Terraform commands:

```
> terragrunt plan
> terragrunt apply
> terragrunt output
> terragrunt destroy
```

Terragrunt forwards almost all commands, arguments, and options directly to Terraform, using whatever version of Terraform you already have installed. However, before running Terraform, Terragrunt will ensure your remote state is configured according to the settings in the *.terragrunt* file. Moreover, for any commands that could change your Terraform state (e.g., *apply* and *destroy*), Terragrunt will acquire and release a lock using DynamoDB.

Here’s what it looks like in action:

```
> terragrunt apply

[terragrunt] Configuring remote state for the s3 backend
[terragrunt] Running command: terraform remote config
[terragrunt] Attempting to acquire lock in DynamoDB
```

```
[terragrunt] Attempting to create lock item table terragrunt_locks
[terragrunt] Lock acquired!
[terragrunt] Running command: terraform apply

terraform apply

aws_instance.example: Creating...
am1: "" => "am1-0d729a60"
instance_type: "" => "t2.micro"

(...)

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

[terragrunt] Attempting to release lock
[terragrunt] Lock released!
```

In this output, you can see that Terragrunt automatically configured remote state as declared in the `terragrunt` file, acquired a lock from DynamoDB, ran `terraform apply`, and then released the lock. If anyone else already had the lock, Terragrunt would have waited until the lock was released to prevent race conditions. Future developers need only check out the repository containing this folder and run `terragrunt apply` to achieve an identical result!

Isolating State Files

With remote state storage and locking, collaboration is no longer a problem. However, there is still one more problem remaining: isolation. When you first start using Terraform, you may be tempted to define all of your infrastructure in a single Terraform file or a set of Terraform files in one folder. The problem with this approach is that all of your Terraform state is now stored in a single file, too, and a mistake anywhere could break everything.

For example, while trying to deploy a new version of your app in staging, you might break the app in production. Or worse yet, you might corrupt your entire state file, either because you didn't use locking, or due to a rare Terraform bug, and now all of your infrastructure in all environments is broken. ⁴

The whole point of having separate environments is that they are isolated from each other, so if you are managing all the environments from a single set of Terraform configurations, you are breaking that isolation. Just as a ship has bulkheads that act as barriers to prevent a leak in one part of the ship from immediately flooding all the others, you should have “bulkheads” built into your Terraform design, as shown in Figure 3-3.

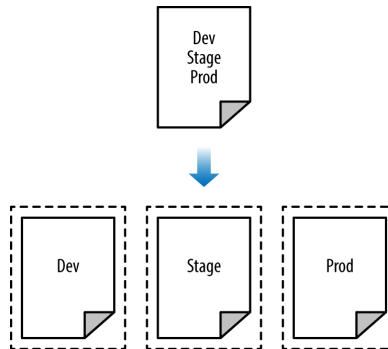


Figure 3-3. Instead of defining all your environments in a single set of Terraform configurations (top), you want to define each environment in a separate set of configurations (bottom), so a problem in one environment is completely isolated from the others.

The way to do that is to put the Terraform configuration files for each environment into a separate folder. For example, all the configurations for the staging environment can be in a folder called `stage` and all the configurations for the production environment can be in a folder called `prod`. That way, Terraform will use a separate state file for each environment, which makes it significantly less likely that a screw up in one environment can have any impact on another.

In fact, you may want to take the isolation concept beyond environments and down to the “component” level, where a component is a coherent set of resources that you typically deploy together. For example, once you’ve set up the basic network topology for your infrastructure—in AWS lingo, your Virtual Private Cloud (VPC) and all the associated subnets, routing rules, VPNs, and network ACLs—you will probably only change it once every few months. On the other hand, you may deploy a new version of a web server multiple times per day. If you manage the infrastructure for both the VPC component and the web server component in the same set of Terraform configurations, you are unnecessarily putting your entire network topology at risk of breakage multiple times per day.

Therefore, I recommend using separate Terraform folders (and therefore separate state files) for each environment (staging, production, etc.) and each component (vpc, services, databases). To see what this looks like in practice, let’s go through the recommended file layout for Terraform projects.

File Layout

Figure 3-4 shows the file layout for my typical Terraform project.

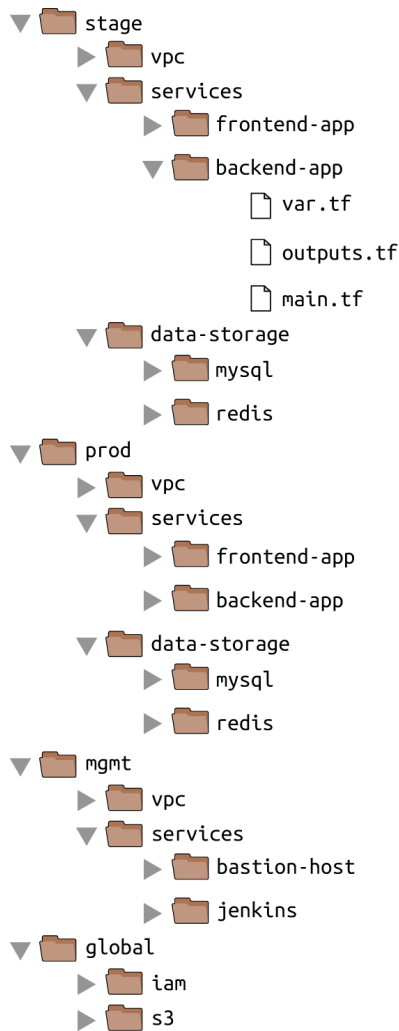


Figure 3-4. Typical file layout for a Terraform project

At the top level, there are separate folders for each “environment.” The exact environments differ for every project, but the typical ones are:

- stage
An environment for nonproduction workloads (i.e., testing).
- prod
An environment for production workloads (i.e., user-facing apps).
- mgmt
An environment for DevOps tooling (e.g., bastion host, Jenkins).
- global
A place to put resources that are used across all environments (e.g., S3, IAM).

Within each environment, there are separate folders for each “component.” The components differ for every project, but the typical ones are:

- vpc
The network topology for this environment.
- services

The apps or microservices to run in this environment, such as a Ruby on Rails frontend or a Scala backend. Each app could even live in its own folder to isolate it from all the other apps.

data-storage

The data stores to run in this environment, such as MySQL or Redis. Each data store could even live in its own folder to isolate it from all other data stores.

Within each component, there are the actual Terraform configuration files, which are organized according to the following naming conventions:

vars.tf

Input variables.

outputs.tf

Output variables.

main.tf

The actual resources.

When you run Terraform, it simply looks for files in the current directory with the *.tf* extension, so you can use whatever filenames you want. Using a consistent convention like this makes your code easier to browse, since you always know where to look to find a variable, output, or resource. If your Terraform configurations are becoming massive, it's OK to break out certain functionality into separate files (e.g., *iam.tf*, *s3.tf*, *database.tf*), but that may also be a sign that you should break your code into smaller modules instead, a topic I'll dive into in [Chapter 4](#).

AVOIDING COPY/PASTE

The file layout described in this section has a lot of duplication. For example, the same *frontend-app* and *backend-app* live in both the *stage* and *prod* folders. Don't worry, you won't need to copy/paste all of that code! In [Chapter 4](#), you'll see how to use Terraform modules to keep all of this code DRY.

Let's take the web server cluster code you wrote in [Chapter 2](#), plus the S3 bucket code you wrote in this chapter, and rearrange it using the folder structure in [Figure 3-5](#).

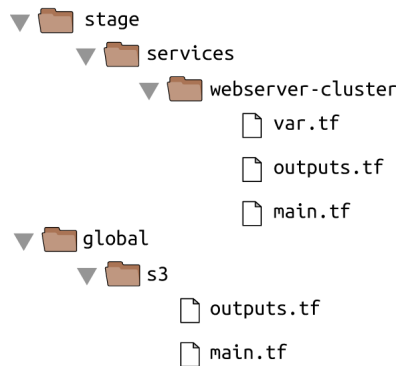


Figure 3-5. File layout for the web server cluster code

The S3 bucket you created in this chapter should be moved into the *global/s3* folder. Note that you'll need to move the *s3_bucket_arn* output variable to *outputs.tf*. If you configured remote state storage, make sure you don't miss the (hidden) *.terraform* folder when copying files to the new location.

The web server cluster you created in [Chapter 2](#) should be moved into *stage/services/webserver-cluster* (think of this as the "testing" or "staging" version of that web server cluster; you'll add a "production" version in the next chapter). Again, make sure to copy over the *.terraform* folder, move input variables into *vars.tf*, and output variables into *outputs.tf*.

You should also configure remote state storage in S3 for the web server cluster (e.g., by running the `terraform remote config` command). Set the S3 key

to the same path as the web server Terraform code: `state/services/webserver-cluster/terraform.tfstate`. This gives you a 1:1 mapping between the layout of your Terraform code in version control and your Terraform state files in S3, so it's obvious how the two are connected.

This file layout makes it easy to browse the code and understand exactly what components are deployed in each environment. It also provides a good amount of isolation between environments and between components within an environment, ensuring that if something goes wrong, the damage is contained as much as possible to just one small part of your entire infrastructure.

Of course, this very same property is, in some ways, a drawback, too: splitting components into separate folders prevents you from breaking multiple components in one command, but it also prevents you from creating all the components in one command. If all of the components for a single environment were defined in a single Terraform configuration, you could spin up an entire environment with a single call to `terraform apply`. But if all the components are in separate folders, then you need to run `terraform apply` separately in each one (note that if you're using Terragrunt, you can automate this process using the `spin-up` command ⁵).

There is another problem with this file layout: it makes it harder to use resource dependencies. If your app code was defined in the same Terraform configuration files as the database code, then that app could directly access attributes of the database (e.g., the database address and port) using Terraform's interpolation syntax (e.g., `${aws_db_instance.foo.address}`). But if the app code and database code live in different folders, as I've recommended, you can no longer do that. Fortunately, Terraform offers a solution: read-only state.

Read-Only State

In Chapter 2, you used data sources to fetch read-only information from AWS, such as the `aws_availability_zones` data source, which returns a list of availability zones in the current region. There is another data source that is particularly useful when working with state: `terraform_remote_state`. You can use this data source to fetch the Terraform state file stored by another set of Terraform configurations in a completely read-only manner.

Let's go through an example. Imagine that your web server cluster needs to talk to a MySQL database. Running a database that is scalable, secure, durable, and highly available is a lot of work. Once again, you can let AWS take care of it for you, this time by using the *Relational Database Service (RDS)*, as shown in Figure 3-6. RDS supports a variety of databases, including MySQL, PostgreSQL, SQL Server, and Oracle.

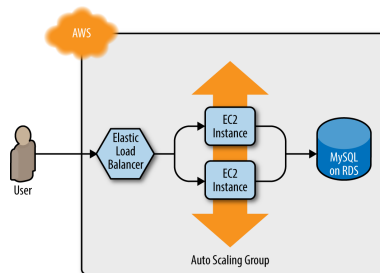


Figure 3-6. The web server cluster talks to MySQL, which is deployed on top of Amazon's Relational Database Service

You may not want to define the MySQL database in the same set of configuration files as the web server cluster, as you'll be deploying updates to the web server cluster far more frequently and don't want to risk accidentally breaking the database each time you do so. Therefore, your first step should be to create a new folder at `stage/data-stores/mysql` and create the basic Terraform files (`main.tf`, `vars.tf`, `outputs.tf`) within it, as shown in Figure 3-7.

Next, create the database resources in `stage/data-stores/mysql/main.tf`:

```

provider "aws" {
  region = "us-east-1"
}

resource "aws_db_instance" "example" {
  engine           = "mysql"
  allocated_storage = 10
  instance_class   = "db.t2.micro"
  name             = "example_database"
  username         = "admin"
  password         = "${var.db_password}"
}

```

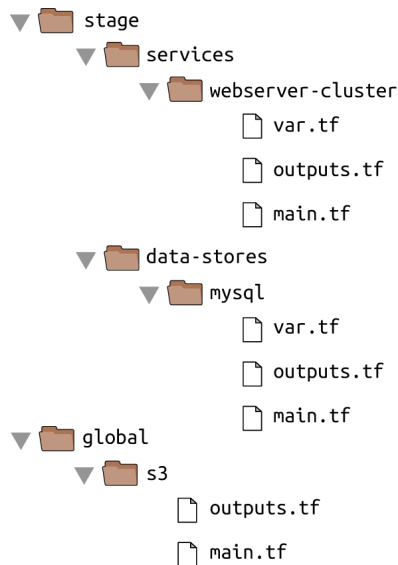


Figure 3-7. Create the database code in the `stage/data-stores` folder

At the top of the file, you see the typical provider resource, but just below that is a new resource: `aws_db_instance`. This resource creates a database in RDS. The settings in this code configure RDS to run MySQL with 10GB of storage on a `db.t2.micro` instance, which has 1 virtual CPU, 1GB of memory, and is part of the AWS free tier. Notice that in this code, the `password` parameter is set to the `var.db_password` input variable, which you should declare in `stage/data-stores/mysql/vars.tf`:

```
variable "db_password" {
  description = "The password for the database"
}
```

Note that this variable does not have a `default`. This is intentional. You should not store your database password or any sensitive information in plain text. Instead, you should store all secrets using a password manager that will encrypt your sensitive data (e.g., 1Password, LastPass, OS X Keychain) and expose those secrets to Terraform via environment variables. For each input variable `foo` defined in your Terraform configurations, you can provide Terraform the value of this variable using the environment variable `TF_VAR_foo`. For the `var.db_password` input variable, here is how you can set the `TF_VAR_db_password` environment variable on Linux/Unix/OS X systems:

```
> export TF_VAR_db_password="(YOUR_DB_PASSWORD)"
```

Next, configure remote state storage so that the database stores all of its state in S3 (e.g., by running the `terraform remote config` command) and set the S3 key to `stage/data-stores/mysql/terraform.tfstate`. As a reminder, Terraform stores all variables in its state files in plain text, including the database password, so make sure to enable encryption when configuring remote state.

Run `terraform plan`, and if the plan looks good, run `terraform apply` to create the database. Note that RDS can take as long as 10 minutes to provision even a small database, so be patient!

Now that you have a database, how do you provide its address and port to your web server cluster? The first step is to add two output variables to `stage/data-stores/mysql/outputs.tf`:

```
output "address" {
  value = "${aws_db_instance.example.address}"
}

output "port" {
  value = "${aws_db_instance.example.port}"
}
```

Run `terraform apply` one more time and you should see the outputs in the terminal:

```
> terraform apply

(...)

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

Outputs:
```

```
address = tf-2016111123.cowu6mts6srx.us-east-1.rds.amazonaws.com
port = 3306
```

These outputs are now also stored in the remote state for the database, which is in your S3 bucket at the path `stage/data-stores/mysql/terraform.tfstate`. You can get the web server cluster code to read the data from this state file by adding the `terraform_remote_state` data source in `stage/services/webserver-cluster/main.tf`:

```
data "terraform_remote_state" "db" {
  backend = "s3"

  config {
    bucket = "(YOUR_BUCKET_NAME)"
    key    = "stage/data-stores/mysql/terraform.tfstate"
    region = "us-east-1"
  }
}
```

This `terraform_remote_state` data source configures the web server cluster code to read the state file from the same S3 bucket and folder where the database stores its state, as shown in Figure 3-8.

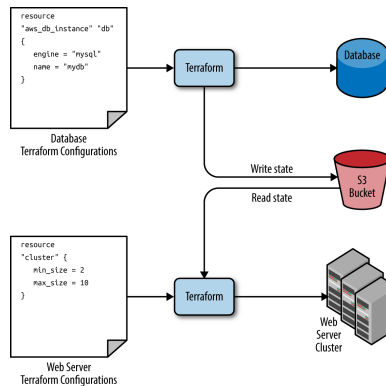


Figure 3-8. The database writes its state to an S3 bucket (top) and the web server cluster reads that state from the same bucket (bottom)

It's important to understand that, like all Terraform data sources, the data returned by `terraform_remote_state` is read-only. Nothing you do in your web server cluster Terraform code can modify that state, so you can pull in the database's state data with no risk of causing any problems in the database itself.

All the database's output variables are stored in the state file and you can read them from the `terraform_remote_state` data source using interpolation syntax:

```
"${data.terraform_remote_state.NAME.ATTRIBUTE}"
```

For example, here is how you can update the User Data of the web server cluster instances to pull the database address and port out of the `terraform_remote_state` data source and expose that information in the HTTP response:

```
user_data = <<EOF
#!/bin/bash
echo "Hello, World" >> index.html
echo "${data.terraform_remote_state.db.address}" >> index.html
echo "${data.terraform_remote_state.db.port}" >> index.html
nohup busybox httpd -f -p "${var.server_port}" &
EOF
```

As the User Data script is getting longer, defining it inline is getting messier and messier. In general, embedding one programming language (Bash) inside another (Terraform) makes it harder to maintain each one, so it's a good idea to externalize the Bash script. To do that, you can use the `file` interpolation function and the `template_file` data source. Let's talk about these one at a time.

An *interpolation function* is a function you can use within Terraform's interpolation syntax:

```
"${some_function(...)}"
```

For example, consider the `format` interpolation function:

```
"${format(FMT, ARGS, ...)}"
```

This function formats the arguments in `ARGS` according to the `sprintf` syntax in the string `FMT`.⁶ A great way to experiment with interpolation functions is to run the `terraform console` command to get an interactive console where you can try out different Terraform syntax, query the state of your infrastructure, and see the results instantly:

```
terraform console
> format("%.3f", 3.14159265359)
3.142
```

Note that the Terraform console is read-only, so you don't have to worry about accidentally changing infrastructure or state!

There are a number of other built-in functions that can be used to manipulate strings, numbers, lists, and maps.⁷ One of them is the `file` interpolation function:

```
"${file(PATH)}"
```

This function reads the file at `PATH` and returns its contents as a string. For example, you could put your User Data script into `stage/services/webserver-cluster/user-data.sh` and load its contents into the `user_data` parameter of the `aws_launch_configuration` resource as follows:

```
user_data = "${file("user-data.sh")}"
```

The catch is that the User Data script for the web server cluster needs some dynamic data from Terraform, including the server port, database address, and database port. When the User Data script was embedded in the Terraform code, you used interpolation syntax to fill in these values. This does not work with the `file` interpolation function. However, it does work if you use a `template_file` data source.

The `template_file` data source has two parameters: the `template` parameter, which is a string, and the `vars` parameter, which is a map of variables. It has one output attribute called `rendered`, which is the result of rendering `template`, including any interpolation syntax in `template`, with the variables available in `vars`. To see this in action, add the following `template_file` data source to `stage/services/webserver-cluster/main.tf`:

```
data "template_file" "user_data" {
  template = "${file("user-data.sh")}"

  vars {
    server_port = "${var.server_port}"
    db_address  = "${data.terraform_remote_state.db.address}"
    db_port     = "${data.terraform_remote_state.db.port}"
  }
}
```

You can see that this code sets the `template` parameter to the contents of the `user-data.sh` script and the `vars` parameter to the three variables the User Data script needs: the server port, database address, and database port. To use these variables, here's what the `stage/services/webserver-cluster/user-data.sh` script should look like:

```
#!/bin/bash

cat > index.html <<EOF
<h1>Hello, World</h1>
<p>DB address: ${db_address}</p>
<p>DB port: ${db_port}</p>
EOF

nohup busybox httpd -f -p "${server_port}" &
```

Note that this Bash script has a few changes from the original:

- It looks up variables using Terraform's standard interpolation syntax, but the only available variables are the ones in the `vars` map of the `template_file` data source. Note that you don't need any prefix to access those variables: e.g., you should use `${server_port}` and not `${var.server_port}`.
- The script now includes some HTML syntax (e.g., `<h1>`) to make the output a bit more readable in a web browser.

A NOTE ON EXTERNALIZED FILES

One of the benefits of extracting the User Data script into its own file is that you can write unit tests for it. The test code can even fill in the interpolated variables by using environment variables, since the Bash syntax for looking up environment variables is the same as Terraform's interpolation syntax. For example, you could write an automated test for `user-data.sh` along the following lines:

```
export db_address=12.34.56.78
export db_port=5555
export server_port=8888

./user-data.sh

output=$(curl "http://localhost:$server_port")

if [[ $output == *"Hello, World"* ]]; then
  echo "Success! Got expected text from server."
else
  echo "Error. Did not get back expected text 'Hello
fi
```

The final step is to update the `user_data` parameter of the `aws_launch_configuration` resource to point to the rendered output attribute of the `template_file` data source:

```
resource "aws_launch_configuration" "example" {
  image_id      = "ami-4bd28157"
  instance_type = "t2.micro"
  security_groups = ["${aws_security_group.instance.id}"]
  user_data     = "${data.template_file.user_data.rendered}"

  lifecycle {
    create_before_destroy = true
  }
}
```

Ah, that's much cleaner than writing Bash scripts inline!

If you deploy this cluster using `terraform apply`, wait for the Instances to register in the ELB, and open the ELB URL in a web browser, you'll see something similar to [Figure 3-9](#).

Yay, your web server cluster can now programmatically access the database address and port via Terraform! If you were using a real web framework (e.g., Ruby on Rails), you could set the address and port as environment variables or write them to a config file so they could be used by your database library (e.g., ActiveRecord) to talk to the database.

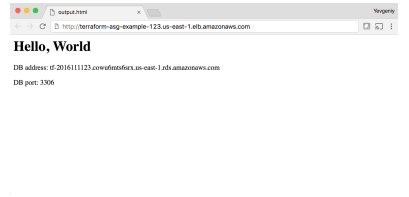


Figure 3-9. The web server cluster can programmatically access the database address and port

Conclusion

The reason you need to put so much thought into isolation, locking, and state is that infrastructure as code (IAC) has different trade-offs than normal coding. When you're writing code for a typical app, most bugs are relatively minor and only break a small part of a single app. When you're writing code that controls your infrastructure, bugs tend to be more severe, as they can break all of your apps—and all of your data stores and your entire network topology and just about everything else. Therefore, I recommend including more "safety mechanisms" when working on IAC than with typical code. ⁸

A common concern of using the recommended file layout is that it leads to code duplication. If you want to run the web server cluster in both staging and production, how do you avoid having to copy and paste a lot of code between `stage/services/webserver-cluster` and `prod/services/webserver-cluster`? The answer is that you need to use Terraform modules, which are the main topic of [Chapter 4](#).

¹ Learn more about S3's guarantees here: <https://aws.amazon.com/s3/details/#durability>.

² See pricing information for S3 here: <https://aws.amazon.com/s3/pricing/>.

³ See here for more information on S3 bucket names: <http://bit.ly/2b1s7eh> (<http://bit.ly/2b1s7eh>).

⁴ For a colorful example of what happens when you don't isolate Terraform state, see: <http://bit.ly/2ITsewM> (<http://bit.ly/2ITsewM>).

⁵ For more information, see [Terraform's documentation](#).

⁶ You can find documentation for the `sprintf` syntax here: <https://golang.org/pkg/fmt/>.

⁷ You can find the full list of interpolation functions here: <https://www.terraform.io/docs/configuration/interpolation.html>.

⁸ For more information on software safety mechanisms, see <http://www.ybrikman.com/writing/2016/02/14/agility-requires-safety/> (<http://www.ybrikman.com/writing/2016/02/14/agility-requires-safety/>).

[Recommended](#) / [Queue](#) / [History](#) / [Topics](#) / [Tutorials](#) / [Settings](#) / [Get the App](#) / [Sign Out](#)

 [PREV](#)
[2. Getting Started with Terraform](#)

[4. How to Create Reusable Infrastructure with Terraform Modules](#) 