What is Terraform state?

When you ran the "terraform apply" command, 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?

The answer is that Terraform 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 templates to the representation of those resources in the real world. For example, let's say your Terraform template contained the following:

```
resource "aws_instance" "example" {
ami = "ami-2d39803a"
instance_type = "t2.micro"
}
```

After running "terraform apply", the terraform.tfstate file will look something like this:

```
"aws_instance.example": {
  "type": "aws_instance",
  "primary": {
    "id": "i-66ba8957",
    "attributes": {
      "ami": "ami-2d39803a",
      "availability_zone": "us-east-1d",
      "id": "i-66ba8957",
      "instance_state": "running",
      "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 (look for the text "Refreshing state…") and compare that to what's in your Terraform templates to determine what changes need to be applied.

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:

1. **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.
2. **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.
3. **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 the staging environment, you want to be sure that you're not going to 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, we'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:

1. **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.
2. **Secrets:** All data in Terraform state files is stored in plaintext. 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, Terraformwill store the username and password for the database in a state file with no encryption whatsoever. Storing plaintext

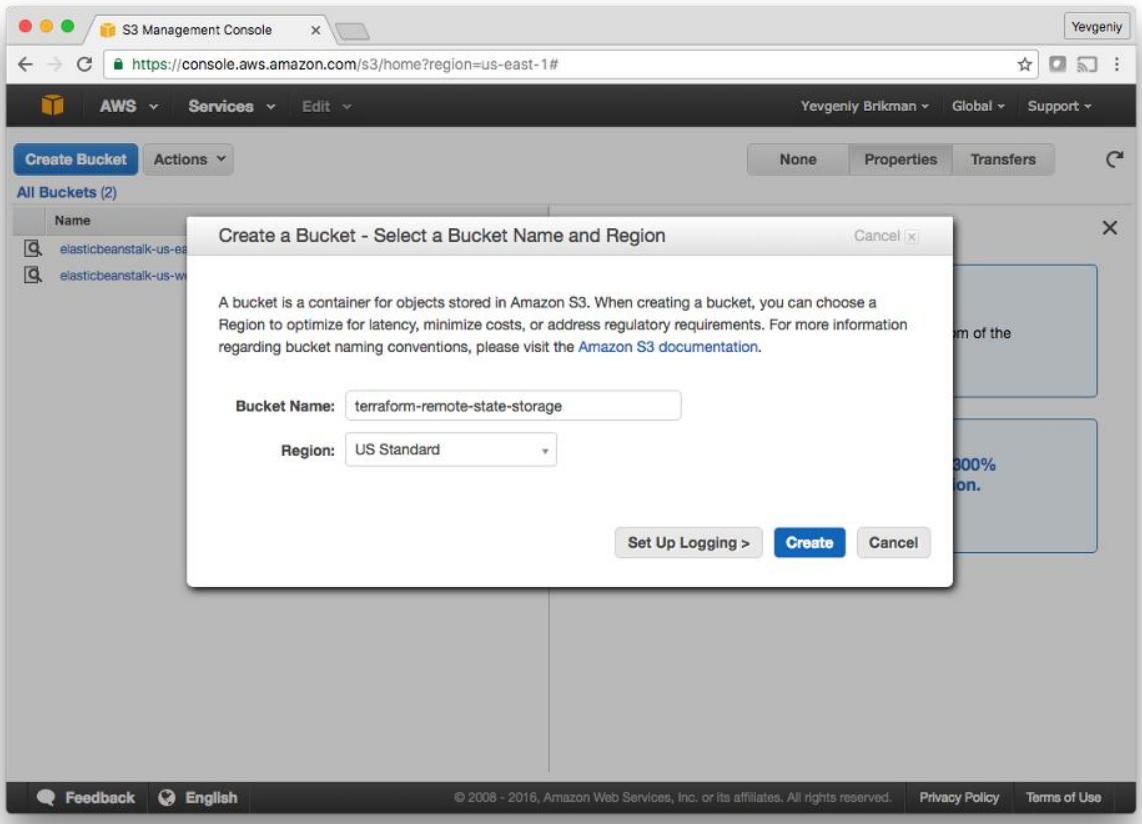
secrets *anywhere* is a bad idea, including version control. This is an open issue in the Terraform community, and we only have partial solutions available at this point, as discussed below.

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, Consul, and HashiCorp Atlas.

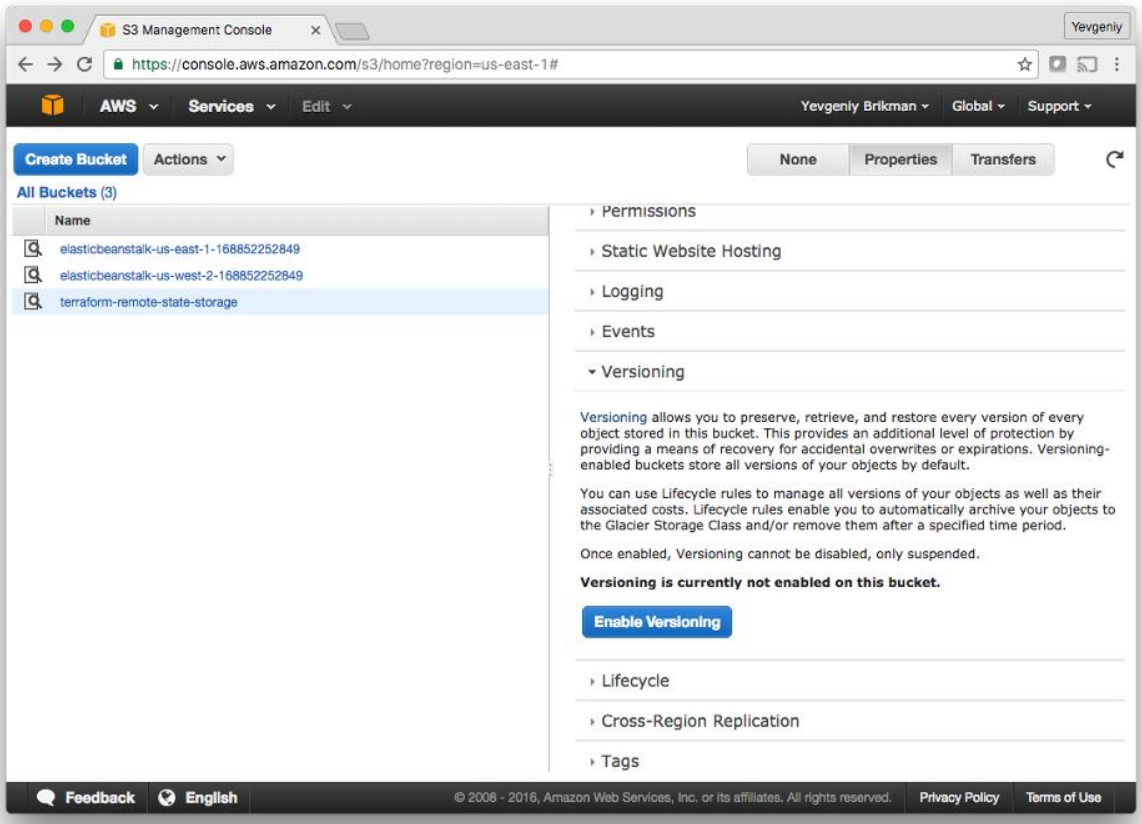
We typically recommend Amazon S3 for the following reasons:

1. It's a managed service, so you don't have to deploy and manage extra infrastructure to use it.
2. It's designed for 99.999999999% durability and 99.99% availability, which effectively means it'll never lose your data or go down.
3. 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, which is not ideal, but at least it's encrypted at rest in S3 and in transit thanks to SSL.
4. It supports versioning, so every revision is stored, and you can always roll back to an older version if something goes wrong.
5. It's inexpensive, with most Terraform usage easily fitting into the free tier.

To enable remote state storage with S3, the first step is to create an S3 bucket. Head over to the S3 console, click the blue "Create Bucket" button, enter a name for the bucket (note: bucket names must be globally unique), and pick a region (you'll need to remember the region for later—note, "US Standard" is us-east-1):



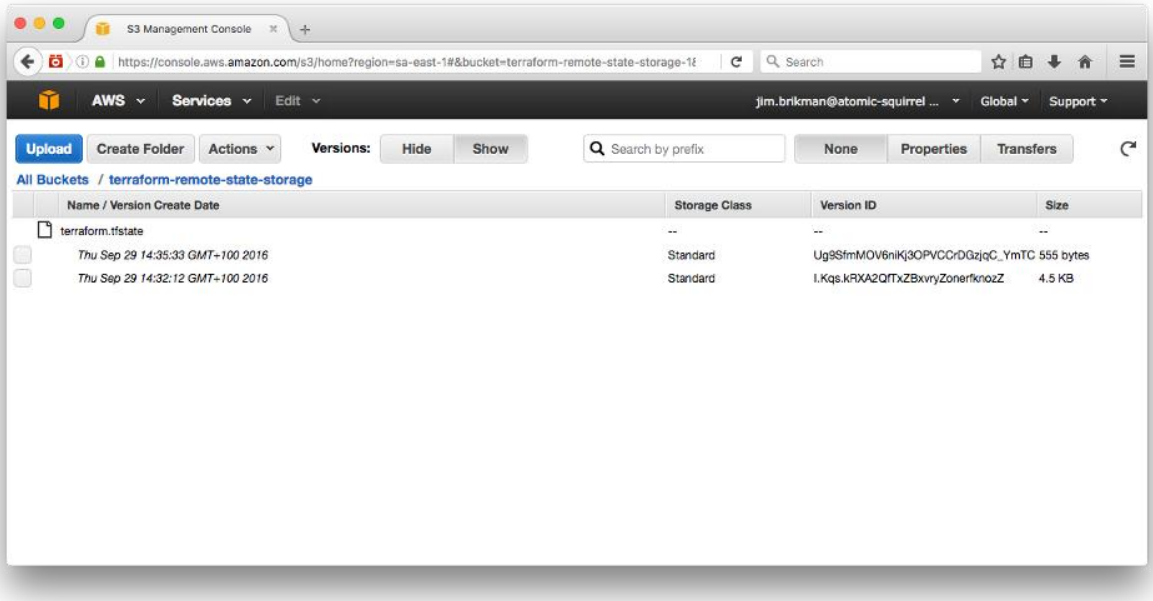
Click the “Create” button and the bucket should show up in your list. Select the bucket and click the “Properties” button in the top right to show the properties you can configure. Expand the “Versioning” section and click the blue “Enable Versioning” button:



To configure Terraform to use this S3 bucket, with encryption enabled, 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=terraform.tfstate" \
    -backend-config="region=(YOUR_BUCKET_REGION)" \
    -backend-config="encrypt=true"
Remote configuration updated
Remote state configured and pulled.
```

After running this command, you should see your Terraform state show up in that S3 bucket (and if you click the “Show” button next to “Versions”, you’ll see all versions of that state):



Locking state files

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. However, there are still two problems:

1. Each developer on your team needs to remember to run the "terraform remote config" command once 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 (unless you are using HashiCorp's paid product Atlas for remote state storage). Therefore, race conditions are still possible if two developers are using Terraform at the same time on the same state files.

To solve this problem, we created an open source tool called Terragrunt. Terragrunt is a thin wrapper for Terraform that manages remote state for you 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.

Install Terragrunt by following the instructions here. Next, create a file called ".terragrunt" in the same folder as your Terraform templates, 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 = "(YOUR_APP_NAME)"
  }
}
# Configure Terragrunt to automatically store tfstate files in S3
remote_state = {
 backend = "s3"
config {
    encrypt = "true"
    bucket = "(YOUR_BUCKET_NAME)"
    key = "terraform.tfstate"
    region = "(YOUR_BUCKET_REGION)"
  }
}
```

The .terragrunt file uses the same language as Terraform, HCL. The first part of the configuration tells Terragrunt to use DynamoDB for locking. The second part of the configuration uses the exact same settings as the "terraform remote config" command to tell Terragrunt to use an S3 bucket for remote state storage.

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

```
terragrunt get
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 the apply and destroy commands, 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…
ami: "" => "ami-0d729a60"
instance_type: "" => "t2.micro"
(...)
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
[terragrunt] Attempting to release lock
[terragrunt] Lock released!
```

In the output above, 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 "git clone" the repo containing this folder and run "terragrunt apply" to achieve an identical result!

You can learn more about Terragrunt's background here and the tech details of how Terragrunt obtains a lock fromDynamoDB here.

Isolating state files

With remote state storage and locking, we've solved the problems of collaboration. However, there is 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 (see here for a colorful example of this happening in the real world).

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 templates, 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.

The way to do that is to put the Terraform templates for each environment into a separate folder. For example, all the templates for the staging environment can be in a folder called "stage" and all the templates 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, we recommend taking 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, network ACLs, etc—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 templates, you are unnecessarily putting your entire network topology at risk of breakage multiple times per day.

Therefore, we recommend using separate Terraform folders (and therefore separate state files) for each environment (staging, production, etc.), and within each environment, each "component." To see what this looks like in practice, let's go through our recommended file layout for Terraform projects.

File layout

Here is the file layout for our typical Terraform project:

```
stage
  └ vpc
  └ services
      └ frontend-app
      └ backend-app
          └ vars.tf
          └ outputs.tf
          └ main.tf
          └ .terragrunt
  └ data-storage
      └ mysql
      └ redis
prod
  └ vpc
  └ services
      └ frontend-app
      └ backend-app
  └ data-storage
      └ mysql
      └ redis
mgmt
  └ vpc
  └ services
      └ bastion-host
      └ jenkins
global
  └ iam
  └ route53
```

At the top level, we have separate folders for each "environment." The exact environments differ for every project, but the typical ones are:

- **stage:** an environment for non-production 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, such as user management (IAM in AWS) and DNS management (Route53in AWS).

Within each environment, we have 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 lives in its own folder so it's isolated from the others.
- **data-storage:** the data stores to run in this environment, such as MySQL or Redis. Each data store lives in its own folder so it's isolated from the others.

Within each component, we have the actual Terraform templates, which we organize according to the following naming conventions:

- **vars.tf:** input variables.
- **outputs.tf:** output variables.
- **main.tf:** the actual resources.
- **.terragrunt:** locking and remote state configuration for Terragrunt.

This file layout makes it easy to browse the code and understand exactly what components are deployed in each environment. It also provides a goodamount 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.

There is, however, one problem with this file layout: it makes it harder to use resource dependencies. If your app code was defined in the same Terraform template as the VPC code, then that app could directly access attributes of the VPC (e.g. subnet IDs) using Terraform's interpolation syntax (e.g. "${aws_subnet.foo.id}"). But if if the app code and VPC code live in different folders, as recommended above, you can no longer do that. Fortunately, Terraform offers a solution: read-only state.

Read-only state

We 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 templates.

For example, let's say you had Terraform templates to create a VPC in the stage/vpc folder:

```
resource "aws_vpc" "stage" {
cidr_block = "10.0.0.0/18"
}
resource "aws_subnet" "public-1a" {
vpc_id = "${aws_vpc.stage.id}"
availability_zone = "us-east-1a"
cidr_block = "10.0.0.0/24"
}
resource "aws_subnet" "public-1b" {
vpc_id = "${aws_vpc.stage.id}"
availability_zone = "us-east-1b"
cidr_block = "10.0.1.0/24"
}
resource "aws_subnet" "public-1c" {
vpc_id = "${aws_vpc.stage.id}"
availability_zone = "us-east-1c"
cidr_block = "10.0.2.0/24"
}
```

This template creates a VPC and several subnets. Now let's say that in the stage/services/backend-service folder, you wanted to create an Elastic Load Balancer (ELB) that ran in those same subnets. How do you get the IDs of those subnets from the VPC templates to the backend-service templates?

The first step is to add an output variable to the VPC templates to export the subnet IDs:

```
output "public_subnet_ids" {
  value = [
    "${aws_subnet.public-1a.id}",
    "${aws_subnet.public-1b.id}",
    "${aws_subnet.public-1c.id}"
  ]
}
```

The next step is to configure the VPC templates to use remote state storage with an S3 bucket by creating a .terragrunt file:

```
# Configure Terragrunt to automatically store tfstate files in S3
remote_state = {
 backend = "s3"
config {
    encrypt = "true"
    bucket = "my-terraform-remote-state"
    key = "stage/vpc/terraform.tfstate"
    region = "us-east-1"
  }
}
```

Note that we explicitly set the “key” (i.e. the folder path in the S3 bucket) to stage/vpc/terraform.tfstate. After you run “terraform apply” on the VPC templates, the state for those templates—including, crucially, that public_subnet_ids output—will be stored in S3.

Now, in your stage/services/backend-service templates, you can use the terraform_remote_state data source to fetch the VPC data from S3 and feed it into your ELB:

```
data "terraform_remote_state" "vpc" {
  backend = "s3"
config {
    bucket = "my-terraform-remote-state"
    key = "stage/vpc/terraform.tfstate"
    region = "us-east-1"
  }
}
resource "aws_elb" "example" {
  name = "example"
  subnets = ["${data.terraform_remote_state.vpc.public_subnet_ids}"]
}
```

The code above configures the terraform_remote_state data source with the exact same settings as the .terragrunt file of the VPC. It then uses Terraform interpolation syntax to pull out the value of the public_subnet_ids output and feed it into the ELB.

It’s important to understand that, like all Terraform data sources, the data returned by terraform_remote_state is read-only. Therefore, you can pull in the VPC’s state data into an app with no risk of causing any problems in the VPC itself.

Conclusion

The reason we 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, we recommend including more "safety mechanisms" when working on IAC than with typical code (for more info, see Agility Requires Safety).