**Manage Terraform State file:**

=====================================================================

While using Terraform to create and update resources, you might 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?

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-0fb653ca2d3203ac1"  
 instance\_type = "t2.micro"  
}

After running terraform apply, here is a small snippet of the contents of the *terraform.tfstate*file (truncated for readability):

{  
 "version": 4,  
 "terraform\_version": "1.2.3",  
 "serial": 1,  
 "lineage": "86545604-7463-4aa5-e9e8-a2a221de98d2",  
 "outputs": {},  
 "resources": [  
 {  
 "mode": "managed",  
 "type": "aws\_instance",  
 "name": "example",  
 "provider": "provider[\"registry.terraform.io/...\"]",  
 "instances": [  
 {  
 "schema\_version": 1,  
 "attributes": {  
 "ami": "ami-0fb653ca2d3203ac1",  
 "availability\_zone": "us-east-2b",  
 "id": "i-0bc4bbe5b84387543",  
 "instance\_state": "running",  
 "instance\_type": "t2.micro",  
 "(...)": "(truncated)"  
 }  
 }  
 ]  
 }  
 ]  
}

Using this JSON format, Terraform knows that a resource with type aws\_instance and name example corresponds to an EC2 Instance in your AWS account with ID i-0bc4bbe5b84387543. 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. In other words, the output of the plan command is a diff between the code on your computer and the infrastructure deployed in the real world, as discovered via IDs in the state file.

***The State File Is a Private API.****The state file format is a private API that 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 or terraform state commands (you’ll see examples of both later in this series).*

If you’re using Terraform for a personal project, storing state in a single *terraform.tfstate*file that lives locally on your computer 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 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](https://www.terraform.io/language/settings/backends/configuration). 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](https://aws.amazon.com/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](https://aws.amazon.com/s3/features/#durability.), which means you don’t need to worry too much about data loss or outages.
* 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.