# Terraform: From Beginner to Master with Examples in AWS

## 1 Your First Terraform Project

### Setting up Your First Project

This lesson will teach you how to set up your first project in Terraform and will go over each part of the project in detail.

In this chapter, we are going to create your first Terraform project. We will not cover everything in great detail as we will circle back and fill in the gaps in later chapters. We want the focus of this chapter to be about getting a feel for running Terraform and actually creating some infrastructure with it.

Setting up your first project

We’ll create a file named main.tf

Inside the file main.tf, we’ll paste the following text (replace <yourname> with your name or something unique):

provider "aws" {

region = "us-east-2"

}

resource "aws\_s3\_bucket" "first\_bucket" {

bucket = "<yourname>-first-bucket"

}

That’s all we need for our first Terraform project. The Terraform code we have just written will create an S3 bucket in AWS with the name <yourname>-first-bucket in the region us-east-2. You may have been able to guess this was going to happen, even without knowing Terraform. That is one of the strong parts of Terraform; the code is very readable and it is normally quite clear what is going to happen.

What does your first project do?

Let us take a second to explain each part of the code in a bit more detail.

First block

In the first 3 lines, we are defining the provider that we want to use. Terraform itself is just an engine that knows how to run a provider that conforms to an interface. The Terraform engine is smart and knows how to create dependency trees and plans, and it uses the provider to interface with the outside world. Since we are going to be using Terraform with AWS in this course, we need to configure the AWS provider.

To configure the provider, we use the keyword provider then follow it with the name of the provider in quotes (in this case "aws"). We start the provider block by opening a curly brace {. We can now specify any parameters we want to configure the provider.

To pass a parameter, you simply put the name of the parameter followed by an equals sign and then the value you want to pass the parameter in quotation markers. In our example, we are setting the region this provider will use to be us-east-2. This is the region where the AWS Terraform provider will create all of the infrastructures we define. We then end the provider block with a closing curly brace }.

Second block

The next block we have defined is a resource. A resource in Terraform represents a thing in the real world (in this case, an S3 bucket). To define a resource, you start a resource block by using the keyword resource. You then follow it with the resource you want to create in quotes. We want to create an S3 bucket, so we are using the S3 resource "aws\_s3\_bucket".

If you are following along in IntelliJ and typing in the code, you might have noticed that IntelliJ gave you a full list of possible resources once you started typing. You can see the full list on the AWS provider page if you are interested. After we have specified the type of resource we want to create, we put another space and then the identifier you want to give that resource in quotes, in our example "first\_bucket".

We then open the block in the same way that we did for the provider block with an opening curly brace {. Next, we can give any parameters the resource takes values. In this case, we are only setting the name of the bucket. We then end the resource block with a closing }.

### Creating Your First Infrastructure With Terraform

Learn to create an infrastructure using Terraform and how Terraform creates and destroys the infrastructure.

The first thing you have to do with a new Terraform project is to initialise Terraform for that project. To do this, provide aws\_access\_key\_id and aws\_secret\_access\_key, change <your-name> for the bucket name, and click the RUN button:

📝Note: Before running the project please do make sure that you have set up an AWS account. You can also refer to the lesson, Setting up an AWS Account.

Important Note:

If you cannot edit the access keys and other fields in the terminal below, you don’t need to provide your credentials. We have got you covered.

Please use only us-east-2 region for working in AWS in this course.

Please ensure that the bucket name is globally unique.

provider "aws" {

region = "us-east-2"

}

resource "aws\_s3\_bucket" "first\_bucket" {

bucket = "<your-name>-first-bucket"

}

Initialise your first Terraform project

You will see some output on the screen as Terraform initialises, then you should see the message Terraform has been successfully initialized!. Once you have initialised Terraform, you are ready to create the infrastructure by running:

terraform apply

Terraform apply command

After you run the apply, you will see quite a lot of output from Terraform. You will notice that the apply has paused and is awaiting a response from you.

Let’s take a second and look at what is happening here. By default, when you run terraform apply, Terraform will look at the code you have written and compare it to the infrastructure you currently have (in this case in AWS). Once Terraform has done this, it calculates a plan.

Terraform plan

The plan is what Terraform will do to get the real infrastructure from where it is now to how you have specified you want it to be in code. From looking at the plan, we can see Terraform is saying it will create an S3 bucket. You have told Terraform you want an S3 bucket, so Terraform went to AWS to check and realised that there is no S3 bucket in AWS with that name. Thus, it knows that it needs to create the bucket.

📝Note: Plans will be discussed in much more detail later in this course.

The great thing about this plan is that Terraform presents it to us and then pauses, giving us time to decide whether we want to go ahead. You can imagine how useful this is if you accidentally make a change that will destroy your database! To get Terraform to make these changes and create the S3 bucket, type yes and press enter.

Terraform apply

Once the apply has finished, you should see the message Apply complete! Resources: 1 added, 0 changed, 0 destroyed.. This is Terraform telling you that it successfully created the S3 bucket for you. Now, let’s work through the following steps:

Log onto the aws console (website). If you go to the S3 section, you will see the bucket that Terraform created.

Delete the bucket from the AWS console. Now go back to the terminal and run terraform apply again.

You will notice that Terraform has worked out that the S3 bucket is not there anymore, so it needs to create it again. At no point did you tell Terraform the bucket was gone, Terraform worked it out on its own.

Confirm the apply (by typing yes), so the S3 bucket exists again.

Now run terraform apply again when the bucket is there. You will see Terraform output Apply complete! Resources: 0 added, 0 changed, 0 destroyed..

Terraform realises that the state of the world is exactly how you want it to be, so Terraform is saying “nothing to do here!”

Terraform destroy

To finish up, let’s destroy the infrastructure we created, don’t worry, Terraform can take care of that for us:

Simply run the command terraform destroy.

Terraform will present a plan to you of what it is going to destroy and then pause so you can confirm.

Type yes and press enter. When the destroy finishes, you will see a message Destroy complete! Resources: 1 destroyed..

This is telling you Terraform has successfully destroyed everything.

Log into the AWS console and go to S3, and you will see that the bucket is now gone.

Summary

That concludes our first experience with Terraform. I hope you can start seeing the power and simplicity that Terraform provides. Feel free to play around with this project and try changing the properties like the name of the S3 bucket and see what happens. That is a great way to learn. Just remember to run terraform destroy when you are finished to ensure that you are not left with any infrastructure running in AWS.

Do not worry if you have more questions about any of the steps we just went through. We are going to cover everything in much more detail soon!

## 2 Resources

### Resources in Detail

This lesson will explore what Terraform resources are, how they work, and how you can add properties to them.

Terraform resources

Resources in Terraform represent things in the real world. For example, a resource could be an AWS Load Balancer, an alarm in PagerDuty, a policy in Vault, etc. The resource is the bedrock of Terraform. They allow you to define how you want to create something in the real world. Remember, you can create resources representing things from multiple vendors (for example, multiple clouds) in a single project.

Your project resource

Let’s take a look in a bit more detail at the resource we defined in the previous chapter:

resource "aws\_s3\_bucket" "first\_bucket" {

bucket = "<yourname>-first-bucket"

}

Resource you defined in your project

The resource type aws\_s3\_bucket starts with the name of the provider followed by an underscore (aws\_). This allows you to tell just from the first word of the resource which vendor or component this resource will be created in.

Resource examples

Let’s take a look at a few other examples:

resource "google\_folder" "department" {

display\_name = "Department"

parent = "organisation/1234567"

}

Resource example

The department resource above will create a folder in Google Cloud (GCP). You can see that it starts with google\_ which is the name of the Google Cloud (GCP) provider. Every resource for this provider will start with google\_.

Postgres provider resource example

resource "postgresql\_role" "my\_role" {

name = "my\_role"

login = true

password = "password123"

}

Postgres provider resource example

The my\_role resource above will create a login on a Postgres database, with login name my\_role and password password123. The resource name starts with postgresql\_ as every resource for the Postgres provider will.

If we look back at our S3 bucket resource, the last word on the line in quotes was "first\_bucket". This is the identifier for that S3 bucket within your Terraform project. The identifier is what we use inside our project to refer to an instance of a resource. If you created multiple instances of the same resource (for example, you could create many S3 buckets), the identifier would provide you a way to reference each one.

Resource properties

The key name value pairs that make up the body of the resource are the properties for the resource. Some properties on the resource are mandatory and some are optional. For an AWS S3 bucket, the only mandatory property is the name of the bucket. We could have set more properties on the bucket, though. For example:

resource "aws\_s3\_bucket" "first\_bucket" {

bucket = "kevholditch-first-bucket"

acl = "private"

versioning {

enabled = true

mfa\_delete = false

}

}

Setting properties for the resource

In the above example, we are setting the acl to private which essentially says that this bucket will only allow private access. We are also setting two properties for versioning, one to say we are enabling versioning and another to say that you do not require MFA to delete an item on this bucket. You will notice that the properties for versioning are nested in another object. This is a design choice by the resource creator that groups all of the versioning properties together. You may also notice that the two properties in the versioning section are booleans (true/false). These do not require quotes around them like strings do, as we have done for the other two properties.

You can get a full list of all of the properties that are supported for an S3 bucket on the Terraform provider documentation page. The documentation is quite easy to find on Google.

Another resource type

Let’s take a look at another resource type to examine the other data types that resources can take in their properties:

resource "aws\_security\_group" "my\_security\_group" {

name = "allow\_tls"

ingress {

protocol = "tcp"

from\_port = 443

to\_port = 443

cidr\_blocks = ["10.0.0.0/16", "11.0.0.0/16"]

}

}

Data resource

In the above resource, we have the two other types of data resources that can take numbers and lists. The port properties (from\_port and to\_port) are numbers and are set by providing a value without quotes. cidr\_blocks is a list type, which takes a list of CIDR blocks to which it will restrict this security group. You can see that a list is given in the same way a JSON array of strings is created, in that you surround it in square braces.

#### GPT

这个 Terraform 代码块是用于在 AWS（亚马逊网络服务）中创建和配置一个安全组。让我们一步一步地了解它的各个部分及其含义，尤其是对于不熟悉网络概念的人来说：

1. \*\*资源类型与名称\*\*:

- `resource "aws\_security\_group" "my\_security\_group"`: 这行代码定义了一个新的资源类型，即 AWS 安全组（Security Group）。`"my\_security\_group"`是我们给这个安全组指定的名称，以便在 Terraform 配置中引用。

2. \*\*安全组名称\*\*:

- `name = "allow\_tls"`: 这里设置了安全组在 AWS 中的显示名称为“allow\_tls”。这个名称有助于识别和管理 AWS 控制台中的安全组。

3. \*\*入站规则\*\*:

- `ingress { ... }`: 这部分定义了安全组的入站规则，也就是控制向安全组内的资源（如 EC2 实例）发送数据的规则。

4. \*\*协议\*\*:

- `protocol = "tcp"`: 指定了协议类型。在这里，它设置为 TCP（传输控制协议），这是一种常用的网络通信协议。

5. \*\*端口范围\*\*:

- `from\_port = 443`

- `to\_port = 443`: 这两行定义了允许数据传输的端口范围。由于起始端口（`from\_port`）和结束端口（`to\_port`）都设置为 443，因此这规则专门允许通过端口 443 的数据。端口 443 通常用于 TLS/SSL 加密的 HTTPS 流量，这是安全的网络通信标准。

6. \*\*CIDR 块\*\*:

- `cidr\_blocks = ["10.0.0.0/16", "11.0.0.0/16"]`: 这指定了一个或多个允许发送到指定端口的 IP 地址范围。CIDR（无类域间路由）块是一种表示 IP 地址范围的方法。在这个例子中，“10.0.0.0/16”和“11.0.0.0/16”代表了两个大型的 IP 地址范围。

总结一下，这个 Terraform 代码块创建了一个名为“allow\_tls”的 AWS 安全组，它允许从两个特定的 IP 地址范围到端口 443（通常用于 HTTPS 流量）的 TCP 流量。这样的配置通常用于确保只有特定的 IP 地址范围可以访问部署在 AWS 上的安全 HTTPS 服务。

#### VPC

CIDR（无类别域间路由）是一种用于表示 IP 地址和其关联网络的方法，广泛应用于网络配置和管理中。理解 CIDR 对于掌握网络基础知识非常重要。让我详细解释一下 CIDR 的概念及其在您提供的 Terraform 示例中的应用：

### CIDR 基础

1. \*\*IP 地址\*\*:

- IP 地址是分配给网络上每个设备的数字标签。在 IPv4（目前最常用的版本）中，它由四组数字组成，每组数字范围从 0 到 255，例如 192.168.1.1。

2. \*\*子网掩码\*\*:

- 子网掩码用于确定 IP 地址中哪些部分表示网络，哪些部分表示主机（即个别设备）。传统的子网掩码，如 255.255.255.0，以点分十进制形式表示。

3. \*\*CIDR 表示法\*\*:

- CIDR 表示法简化了子网掩码的表示。例如，`192.168.1.0/24` 表示 IP 地址范围 `192.168.1.0` 到 `192.168.1.255`。`/24` 表示前 24 位是网络地址，剩下的位数（在这种情况下是 8 位）用于主机地址。

### CIDR 在 Terraform 示例中的应用

在您的 Terraform 配置中，CIDR 用于定义安全组规则，指定哪些 IP 地址范围可以访问特定的端口。例如：

- `cidr\_blocks = ["10.0.0.0/16", "11.0.0.0/16"]`

这里，有两个 CIDR 块：

1. \*\*10.0.0.0/16\*\*:

- 这表示 10.0.0.0 到 10.0.255.255 的所有 IP 地址。`/16` 意味着前 16 位是网络地址，因此这个范围包含了 65,536（2^16）个可能的 IP 地址。

2. \*\*11.0.0.0/16\*\*:

- 同样地，这表示从 11.0.0.0 到 11.0.255.255 的 IP 地址，也包括 65,536 个地址。

在这个 Terraform 示例中，CIDR 块用于指定允许通过 TCP 协议在 443 端口（通常用于 HTTPS）进行通信的 IP 地址范围。这意味着任何在这两个 IP 范围内的设备都被允许访问通过该安全组保护的 AWS 资源，如 EC2 实例，前提是访问请求是通过 443 端口发起的。这样的设置有助于限制访问，增强网络安全性。

### Interpolation Syntax

In this lesson, we'll discuss the attributes a resource returns and will go through an example project to understand attributes reference.

Useful resource output

Once a resource is created, it returns a number of attributes. The attributes a resource returns can be found in the “Attributes Reference” section on the documentation page for any resource. This is amazingly useful as it allows you to use the output from one resource as the argument for another resource.

Project example

Consider the following project:

provider "aws" {

region = "us-east-2"

}

resource "aws\_vpc" "my\_vpc" {

cidr\_block = "10.0.0.0/16"

}

resource "aws\_security\_group" "my\_security\_group" {

vpc\_id = aws\_vpc.my\_vpc.id

name = "Example security group"

}

resource "aws\_security\_group\_rule" "tls\_in" {

protocol = "tcp"

security\_group\_id = aws\_security\_group.my\_security\_group.id

from\_port = 443

to\_port = 443

type = "ingress"

cidr\_blocks = ["0.0.0.0/0"]

}

Create AWS VPC with CIDR block using this project

This project creates an AWS VPC with CIDR block 10.0.0.0/16. Then it defines a security group (aws\_security\_group). Notice that the value of vpc\_id is set to aws\_vpc.my\_vpc.id in the definition of the security group . The value of aws\_vpc.my\_vpc.id is not known before we run the project, since AWS will randomly assign it when we create the VPC. By referencing the VPC we created, we can use this value even though we do not know what it will be until we run the project.

Output attribute

The format for using an output attribute from a resource is <resource\_type>.<resource\_identifier>.<attribute\_name>. In the VPC id example, we are getting the output from an aws\_vpc resource type, with the identifier name my\_vpc, and we want to get the id attribute value. Thus, we end up with aws\_vpc.my\_vpc.id. It is worth noting here that this syntax was greatly simplified in Terraform version 0.15 (which is the syntax all of the examples in this course will be using).

Security group rule

Next, in our project, we define a security group rule (aws\_security\_group\_rule). This allows ingress traffic on port 443. In the aws\_security\_group\_rule, we need to reference the id of the security group that we want to put this rule in. We can use the same technique as we did when we referenced the id of the VPC. Let’s work through how to figure this out together:

It will start with the type of resource we want to reference, aws\_security\_group.

Next, we use the identifier to specify which instance of the security group we want to use, which is my\_security\_group.

Lastly, we use the attribute of that property we want to use, which is id.

This leads us to build the expression aws\_security\_group.my\_security\_group.id which we can use for the value of the property security\_group\_id inside the aws\_security\_group\_rule resource.

Adding a new security group

To illustrate the way Terraform can create a project in parallel, consider what happens when we add a new security group rule to our project above:

resource "aws\_security\_group\_rule" "http\_in" {

protocol = "tcp"

security\_group\_id = aws\_security\_group.my\_security\_group.id

from\_port = 80

to\_port = 80

type = "ingress"

cidr\_blocks = ["0.0.0.0/0"]

}

New security rule group

When you run the project now, Terraform will realise that it can create both security group rules in parallel. Once the security group they both depend on is created, it will create both of the rules together. This feature of Terraform makes for excellent performance. It may seem obvious in this example, but as a project grows, the amount Terraform can run in parallel can be quite impressive.

📝Note: Once you are done with the project, do not forget to run terraform destroy and then confirm with yes to delete all of the resources in this project.

## 3 Providers

### Providers in Detail

This lesson will cover what Terraform providers are and how they work.

Terraform providers

A provider in Terraform is a connection that allows Terraform to manage infrastructure using a pre-defined interface. This abstraction means the provider is completely separate from the Terraform engine - which understands how to read state from a provider, read HCL code, and work out how to get to the desired state. This allows you to write a provider to connect to anything that has a programmable way to talk to it. All the writer of a provider has to do for each resource they want Terraform to control is provide Terraform with a way to create it, read it, and delete it.

📝Note: Updates are actually optional as Terraform can always delete and then create the resource if the update is not provided.

A provider in Terraform is a connection that allows Terraform to manage infrastructure using a predefined interface

Provider model

Due to the provider model that Terraform employs, providers are not part of the main Terraform source code. They are separate binaries that live in their own repositories and can move at their own speed. This means that if a provider needs to release a bug fix or new feature, they can release it. They do not need to coordinate a release of the main Terraform code base.

Provider examples

A provider is defined using a provider block. You have already used a provider block in the examples covered so far in this course. A provider block is shown below:

provider "aws" {

region = "us-east-2"

}

Provider block

The provider block is very simple. It starts with the keyword “provider” to indicate that this is a provider block. You then have to give the name of the provider that you are using. In this case we are using the AWS provider, so we put "aws". You then use a { to open the provider block. You can put all of the configurations you want for the provider inside the provider block. For the AWS provider, the only property that we are configuring is the region. This will be the region in which we are going to create our AWS resources. You then end the provider block with a closing }.

Downloading AWS provider

If you connect terminals and run terraform init, you will notice that Terraform downloads the AWS provider automatically. It will print the following message: Downloading plugin for provider "aws" (hashicorp/AWS) 2.27.0.... How does this happen, though? Hashicorp (the company that makes Terraform) hosts a registry that contains the most popular providers. If the provider you use is in the registry, then all you have to do to use it is define a provider block that sets up that provider and run terraform init to download it. You can see a full list of providers in the registry on the Hashicorp Terraform site.

Where Terraform puts the provider

What happens to the provider then? Terraform simply puts it inside the project where you are currently working in a special folder called .terraform.

The provider is actually a separate binary that Terraform calls out at a run time to do its work. As an interesting aside, the name of the provider binary is always in the format terraform-provider-<NAME>\_vX.Y.Z. Terraform uses this convention to search for providers on your machine so that it knows if you have a particular version of a provider when you run terraform init. Based on this, it decides whether or not to download it.

### Provider Best Practices

An overview of some tips and tricks for Terraform providers.

When you ran terraform init, Terraform downloaded the latest versions of all of the providers that you used in your project. You may have noticed the warning message that Terraform printed out:

Terraform has created a lock file .terraform.lock.hcl to record the provider

selections it made above. Include this file in your version control repository

so that Terraform can guarantee to make the same selections by default when

you run "terraform init" in the future.

terraform init warning message

Terraform creates a special file that is used to record the version of all of the providers used. This is so that when terraform init is run again (potentially on another machine), Terraform can select exactly the same versions of the providers that were used when the project was first to run. This is important for repeatability.

Terraform creates a special file that is used to record the version of all of the providers used

It can be useful to control the version of the provider that Terraform selects when init is first run. For example, some of your code may only work with v3 of a provider because there is a breaking change in v4. By committing the .terraform.lock.hcl file as suggested in the message, you will guarantee that the version of the provider you used when you set up this project will always be used.

Terraform project example

If you want to instruct Terraform to select a certain version of a provider when it first initializes (in case there is no lock file present) we can use a special terraform block to configure the version for the AWS provider.

Let’s see an example of this:

provider "aws" {

region = "us-east-2"

}

terraform {

required\_providers {

aws = {

version = "~> 3.46"

}

}

}

main.tf file of our Terraform providers project example

In the Terraform block above, we have defined a required\_providers block. This required provider block allows us to specify extra properties for each of the providers we are using in the project. Under required providers, we open an AWS block. There, we specify a version constraint for the AWS provider.

Running the project

Clicking the terminal will run terraform init. If prompted, enter yes to run the project.

The version constraint ∼> 3.46 is used by Terraform when there is no .terraform.lock.hcl present. This constraint means that any version greater than or equal to 3.46 but less than 4.0 is allowed. In normal operation, you should not need to worry about version constraints as long as you commit the lock file.

### More Than One Instance of the Same Provider

We'll cover the following

Creating multiple instances of the same provider

Project example

AWS provider instances

VPC creation

Provider property

Project output

Running the project

We will learn to create multiple instances of the same provider and to add provider property.

Creating multiple instances of the same provider

Since region is a required parameter in the AWS provider, you may be wondering how to create resources in different regions. Is that possible in a single Terraform project? Well, yes, it is. To do this, you simply create multiple instances of the same provider.

Project example

Consider the following project:

provider "aws" {

region = "us-east-1"

}

provider "aws" {

region = "us-east-2"

alias = "ohio"

}

resource "aws\_vpc" "n\_virginia\_vpc" {

cidr\_block = "10.0.0.0/16"

}

resource "aws\_vpc" "ohio\_vpc" {

cidr\_block = "10.1.0.0/16"

provider = aws.ohio

}

AWS provider instances

As you can see, we are defining two instances of the AWS provider. One points at the region us-east-1 while the other points at the region us-east-2. For the one that is pointing at us-east-2, you will notice that we are setting the alias property to ohio. Alias is a property you can set on any provider block. It is in no way special to the AWS provider. What this gives us is a way to distinguish between the two providers. Once you define two or more instances of the same provider, every definition must have an alias set after the first.

VPC creation

After we have defined the two AWS providers, we will create a VPC called n\_virginia\_vpc with the CIDR block 10.0.0.0/16. As we have not told Terraform which provider instance to use for this resource, Terraform will pick the instance of the AWS provider where you have not defined an alias. This means that this VPC will be created in the region us-east-1 (N. Virginia). The second VPC we have defined with the identifier ohio\_vpc has a CIDR block of 10.1.0.0/16. This time we have set the provider property on the resource to aws.ohio. This means that Terraform will use our second AWS provider, which points to the region us-east-2. When we run the project by doing terraform apply, this VPC will be created in us-east-2.

Provider property

Every resource has a provider property that you can set. The format of the value is set by <provider\_- name>.<provider\_alias>. For our example, we were using the AWS provider with the alias “Ohio” when we created the VPC in Ohio. Thus, we set the provider property to aws.ohio.

You can also explicitly tell Terraform to use the default provider. To do this, you can set the provider property to the provider name. So, for example, we could have defined the n\_virginia\_vpc:

resource "aws\_vpc" "n\_virginia\_vpc" {

cidr\_block = "10.0.0.0/16"

provider = "aws"

}

Setting provider property to the provider name

Project output

This would have given us the same output, thus adding extra code with no added value. No one in the community writes their HCL as a result. Less is more, and it is cleaner and easier to read if you omit the provider property when using the default provider instance.

Running the project

After running the project, it will create two VPCs, one in us-east-1 and one in us-east-2. Log into the AWS Console and navigate to find the VPC section. Switch into the us-east-1 and us-east-2 regions and you will see the VPCs that Terraform created.

📝Note: Once you are done with the project, do not forget to run terraform destroy and then confirm with yes to delete all of the resources in this project.

## 4 Data Sources

### Data Sources in Detail

This lesson will teach you about Terraform data and how it is to used.

We'll cover the following

Terraform data source

Project example

Referencing a bucket

AWS IAM policy

S3 bucket data source usage

Terraform data source

A data source in Terraform is used to fetch data from a resource that is not managed by the current Terraform project. This allows it to be used in the current project. You can think of it as a read-only resource that already exists; the object exists, but you want to read specific properties of that object for use in your project.

A data source in Terraform is used to fetch data from a resource that is not managed by the current Terraform project

Project example

Let’s dive into an example:

provider "aws" {

region = "us-east-2"

}

data "aws\_s3\_bucket" "bucket" {

bucket = "kevholditch-already-exists"

}

resource "aws\_iam\_policy" "my\_bucket\_policy" {

name = "my-bucket-policy"

policy = <<EOF

{

"Version": "2012-10-17",

"Statement": [

{

"Action": [

"s3:ListBucket"

],

"Effect": "Allow",

"Resource": [

"${data.aws\_s3\_bucket.bucket.arn}"

]

}

]

}

EOF

}

main.tf file of data source project example

As you can see from the above project, a data source block starts with the word “data”. The next word is the type of data source. We are using an aws\_s3\_bucket data source, which is used to lookup an S3 bucket. After the data source type, we give the data source an identifier, in this case "bucket". The identifier is used to reference the data source inside the Terraform project. The data source block is opened with a {. You then specify any properties you want Terraform to use to search for the resource. We are using the complete name of the S3 bucket we are looking for. You then close the data source block with }.

Referencing a bucket

Rather than creating the bucket as we did before, here we are referencing a bucket that already exists. So, before you run the above project, you will need to create an S3 bucket with the name that you specify inside the data block. In the example above, the bucket would be called kevholditch-already-exists. Name the bucket anything you want, but make sure to paste the name into the bucket property in the data source.

AWS IAM policy

At the bottom of this project, we create an AWS IAM policy that gives permissions to list the bucket that we looked up in the data source. There are a couple of new concepts in the aws\_iam\_policy resource that we first need to introduce. The IAM policy itself is a multi-line string enclosed in between <<POLICY and POLICY. This is how you define a multi-line string in Terraform. You open the multi-line string with << then place any identifier you wish as a single word. I have used POLICY in the example above since we are defining an IAM policy, but you can use any label like <<STATEMENT or <<IAM. You then start your multi-line string on the next line and finish it with the opening identifier without the <<.

📝Note: The closing marker must be at the start of a new line. Otherwise, it is a syntax error.

S3 bucket data source usage

Inside the IAM policy, we are using the S3 bucket data source. We take the arn from the S3 bucket so that we can use it in our IAM policy. You will notice that to get the value, we use the interpolation syntax ${data.aws\_s3\_bucket.bucket.arn}. The opening ${ and closing } are necessary because we are inside a multi-line string. They tell terraform that we want it to evaluate this value and not use it as a string literal. The format of a data source expression is data.<data\_type>.<data\_identifier>.<attribute\_name>. You can get a full list of the attributes that a data resource provides from the documentation website of the provider.

We are taking the arn from the S3 bucket so that we can use it in our IAM policy

#### GPT

这个 Terraform（TF）文件是用于配置 AWS（亚马逊网络服务）资源的。让我们一步步详细解读这个文件中的各个部分：

### 1. AWS 提供者配置

```hcl

provider "aws" {

region = "us-east-2"

}

```

- `provider "aws"`: 这一部分指定了 Terraform 将使用 AWS 作为云服务提供者。

- `region = "us-east-2"`: 这里设置 AWS 的地区为“us-east-2”，即美国东部（俄亥俄州）地区。这意味着所有通过这个 Terraform 文件创建的资源将位于这个特定地区。

### 2. 数据源配置

```hcl

data "aws\_s3\_bucket" "bucket" {

bucket = "kevholditch-already-exists"

}

```

- `data "aws\_s3\_bucket" "bucket"`: 这一部分定义了一个数据源，用于引用一个已经存在的 AWS S3 存储桶。

- `bucket = "kevholditch-already-exists"`: 这里指定了存储桶的名称为“kevholditch-already-exists”。通过这个数据源，Terraform 可以获取这个存储桶的相关信息，如其 Amazon Resource Name（ARN）。

### 3. IAM 策略资源

```hcl

resource "aws\_iam\_policy" "my\_bucket\_policy" {

name = "my-bucket-policy"

policy = <<EOF

{

"Version": "2012-10-17",

"Statement": [

{

"Action": [

"s3:ListBucket"

],

"Effect": "Allow",

"Resource": [

"${data.aws\_s3\_bucket.bucket.arn}"

]

}

]

}

EOF

}

```

- `resource "aws\_iam\_policy" "my\_bucket\_policy"`: 这部分定义了一个新的 AWS IAM（身份与访问管理）策略资源。

- `name = "my-bucket-policy"`: 为这个 IAM 策略指定了一个名字，即“my-bucket-policy”。

- `policy = <<EOF ... EOF`: 这里定义了 IAM 策略的具体内容。采用了多行字符串（heredoc）语法来定义 JSON 格式的策略。

- `"Version": "2012-10-17"`: 指定了策略语言的版本。

- `"Statement": [...]`: 策略声明部分，定义了具体的权限。

- `"Action": ["s3:ListBucket"]`: 指定了允许的操作，这里是列出 S3 存储桶的内容。

- `"Effect": "Allow"`: 设置了这个操作的效果为允许。

- `"Resource": ["${data.aws\_s3\_bucket.bucket.arn}"]`: 指定了这个策略适用的资源，这里是上面定义的 S3 存储桶的 ARN。

总体来说，这个 Terraform 文件配置了 AWS 的地区，引用了一个已存在的 S3 存储桶，并创建了一个 IAM 策略，允许列出该存储桶的内容。这个配置有助于在 AWS 环境中管理资源访问和权限。

这个 `aws\_iam\_policy` 定义的是一个基于角色的策略（Identity-based policy），而不是基于资源的策略（Resource-based policy）。

### 基于角色的策略（Identity-based policy）

- \*\*定义\*\*: 这类策略是直接附加到 IAM 用户、用户组或角色上的。它们用于定义这些身份（用户、组或角色）可以执行哪些操作，以及在什么条件下可以执行这些操作。

- \*\*本例中\*\*: `aws\_iam\_policy` 资源创建了一个 IAM 策略。这个策略目前还没有与任何特定的 IAM 用户、组或角色关联。为了使其生效，你需要将其附加到某个 IAM 实体上。

### 基于资源的策略（Resource-based policy）

- \*\*定义\*\*: 基于资源的策略是直接附加到资源上的，例如 S3 存储桶、SQS 队列等。这类策略定义谁可以执行哪些操作以及在什么条件下可以对这些资源执行操作。

- \*\*与本例的区别\*\*: 如果策略是直接附加到 S3 存储桶等资源上，那么它就是基于资源的策略。在你提供的代码中，策略是作为独立的 IAM 策略创建的，而不是直接附加到某个 AWS 资源上。

因此，根据你的 Terraform 代码，创建的是一个可用于附加到 IAM 用户、组或角色的基于角色的策略。

你是对的。在您提供的 Terraform 代码中，虽然定义了一个 IAM 策略 (`aws\_iam\_policy`)，但并没有显示这个策略被附加到任何 IAM 用户、角色或组上。仅仅定义一个 IAM 策略并不会自动将它应用到任何 AWS 身份（用户、角色或组）上。

要使这个策略生效，您需要进行额外的步骤，将这个策略附加到特定的 IAM 用户、角色或组。在 Terraform 中，这通常通过使用 `aws\_iam\_policy\_attachment`、`aws\_iam\_user\_policy\_attachment`、`aws\_iam\_role\_policy\_attachment` 或 `aws\_iam\_group\_policy\_attachment` 资源来完成。

例如，如果您想将这个策略附加到一个 IAM 角色上，您需要添加类似以下的 Terraform 配置：

```hcl

resource "aws\_iam\_role\_policy\_attachment" "my\_attachment" {

role = "指定的IAM角色名称"

policy\_arn = aws\_iam\_policy.my\_bucket\_policy.arn

}

```

这里，`aws\_iam\_role\_policy\_attachment` 资源创建了一个策略附加，将您定义的策略 (`aws\_iam\_policy.my\_bucket\_policy`) 附加到指定的 IAM 角色上。`role` 属性是您想附加策略的角色的名称，`policy\_arn` 是您创建的策略的 ARN（Amazon 资源名称），这可以通过 Terraform 的资源引用直接获取。

这个策略代码是 JSON 格式的 AWS IAM 策略语法，嵌入在 Terraform（TF）配置文件中。

### 解析策略代码的格式和语法

1. \*\*AWS IAM 策略语法（JSON格式）\*\*:

- IAM 策略通常以 JSON 格式编写，遵循 AWS 的策略语法。

- 策略定义了哪些操作（Actions）被允许或拒绝，对哪些资源（Resources）有效，以及在什么条件下（Conditions）这些规则适用。

2. \*\*在 Terraform 配置中的应用\*\*:

- Terraform 使用 HCL（HashiCorp 配置语言）语法编写，但它也可以嵌入其他数据格式，比如 JSON。

- 在您的 Terraform 代码中，IAM 策略被定义在一个多行字符串（使用 `<<EOF ... EOF`）内，这是 Terraform 的一种处理多行或嵌入式文本的方法。

### 具体例子中的应用

在您提供的 Terraform 代码中，IAM 策略被定义在 `aws\_iam\_policy` 资源中：

```hcl

resource "aws\_iam\_policy" "my\_bucket\_policy" {

name = "my-bucket-policy"

policy = <<EOF

{

"Version": "2012-10-17",

"Statement": [

{

"Action": [

"s3:ListBucket"

],

"Effect": "Allow",

"Resource": [

"${data.aws\_s3\_bucket.bucket.arn}"

]

}

]

}

EOF

}

```

- 这里，`policy` 部分是标准的 AWS IAM 策略，采用 JSON 格式编写。

- Terraform 使用 `<<EOF ... EOF` 来定义多行字符串，其中包含 JSON 格式的 IAM 策略。

总结来说，策略代码本身遵循 AWS IAM 策略的 JSON 格式，而这段代码被嵌入在遵循 HCL 语法的 Terraform 配置文件中。

### How are Data Sources Useful?

In this lesson, we'll discuss Terraform data source advantages.

We'll cover the following

Terraform data source advantages

Example

Another advantage

Terraform data source advantages

As your Terraform project grows, it can be sensible to break it up into smaller projects to make for easier maintenance. When this happens, you can use data sources to reference and utilize resources across Terraform projects. In this case, it would always be better to use a data source rather than to compute the arn yourself, as with something like an S3 bucket. This is because you want Terraform to fail if, for some reason, the bucket no longer exists. By using a data source, you maintain this behavior.

As your Terraform project grows it can be sensible to break it up into smaller projects to make for easie maintainance

Example

Imagine you want to create a new AWS EC2 instance using an AMI image from a private repository.

You could hard code the name of the AMI image when creating the instance and manually update it when a new AMI image is released.

This would technically work, but is quite cumbersome and would require a code change every time you wanted to use the latest version of the AMI image.

By using a data source instead, you can set things up to always read the repository and retrieve the latest version of the AMI image when you run Terraform.

You can then reference the data source when creating the EC2 instance to ensure that you always have the latest version of the image.

Another advantage

Another case when you may want to use a data source is if you are migrating existing infrastructure to Terraform and want to reference a resource that is not part of your Terraform project yet. As previously stated, it is always better to use a data source rather than compute the value yourself. You want Terraform to know that there is a dependency on the resource so that your terraform apply will fail if the resource cannot be found or if the attribute it returns changes. Terraform will realise this when you run “apply” and update your project with the new value.

Another use for data sources is when you are migrating existing infrastructure to Terraform and want to reference a resource that is not part of your Terraform project yet.

## 5 Outputs

### Outputs Explained

In this lesson, you'll learn how to write an output in Terraform, how they work, and how they are defined.

We'll cover the following

Terraform project output

Terraform project output example

Let’s dive into the project output

Output definition in Terraform

Terraform project output

Output in your Terraform project shows a piece of data after Terraform successfully completes. Outputs are useful as they allow you to echo values from the Terraform run to the command line. For example, if you are creating an environment and setting up a bastion jump box as part of that environment, it’s handy to be able to echo the public IP address of the newly created bastion to the command line. Then, after the terraform apply finishes, you are given the IP of the newly created bastion which is ready for you to ssh straight onto it.

Terraform project output example

Let’s start with an example of outputs:

output "message" {

value = "Hello World"

}

Terraform project example

Try running this project by connecting with the terminal. After running terraform init and terraform apply. You will see that Terraform runs and then prints the following:

Let’s dive into the project output

You can see from the message above that Terraform states that nothing changed (0 added, 0 changed, 0 destroyed.). You then see Outputs: under which Terraform prints out the values of all of the outputs you have defined. We defined a single output with the identifier message and gave it the value Hello world, so that is what Terraform printed.

Output definition in Terraform

To define an output, you open an output block by using the output keyword. You then start the output block with {. You are only allowed to set a single property called value. Whatever value you give to the value property will be outputted to the console after a successful terraform apply. You then close the output block with }.

📝Note: Outputs are used in modules too and have slightly different semantics. This will be covered in the chapter on modules.

### Outputting Resource Properties

Learn more about Terraform output and output resource properties.

We'll cover the following

Terraform output project example

Project output

The first example is pretty basic and, in the real world, probably not very useful. Outputs are much more useful when used to output the values of resources that have been created as part of a Terraform run.

Terraform output project example

Let’s create another Terraform project and output the values of resources as a part of a Terraform run:

provider "aws" {

region = "us-east-2"

}

resource "aws\_s3\_bucket" "first\_bucket" {

bucket = "kevholditch-bucket-outputs"

}

output "bucket\_name" {

value = aws\_s3\_bucket.first\_bucket.id

}

output "bucket\_arn" {

value = aws\_s3\_bucket.first\_bucket.arn

}

output "bucket\_information" {

value = "bucket name: ${aws\_s3\_bucket.first\_bucket.id}, bucket arn: ${aws\_s3\_bucket.first\_bucket.arn}"

}

main.tf file of Terraform output project example

Let’s walk through the above code.

The provider and resource should be familiar to you. We are simply defining the AWS provider to be used with the us-east-2 region and setting up an S3 bucket.

Feel free to change the name of the bucket to whatever you wish.

Next, we define an output called bucket\_name.

In the bucket\_name, we are going to output the name of the bucket by using the attribute of the S3 bucket resource that we create.

We use the same technique to output the ARN of the bucket that we will create in the output bucket\_arn.

Because we are directly using the attribute in both of those examples, we can just set it equal to value without any quotes.

The last output bucket\_information prints an interpolated string which will give us the bucket name and bucket arn. Since this value is a string with interpolated values, we have to surround it in quotes and ${ }.

Project output

Clicking the terminal will run terraform init and then terraform apply. When prompted, enter yes to run the project. Terraform runs, creates the S3 bucket and gives the following output under the Outputs: heading:

Running the Terraform project

Terraform got the values from the S3 bucket that it created and outputted them when the run completed. Note that Terraform prints the outputs in alphabetical order, not the order that you define them in your project.

Terraform does not care which order you define the blocks in your project. Try reordering them and running terraform apply again. You will notice that Terraform will say that there is nothing to do.

### Exporting all Attributes

Learn to output an entire resource or data block using the Terraform output resource.

We'll cover the following

Running the project

As of Terraform version 1.0.0 (which this course is based on), Terraform allows you to output an entire resource or data block. To do this, we’ll take the example that we just had in the last lesson and added the output "all" block:

provider "aws" {

region = "us-east-2"

}

resource "aws\_s3\_bucket" "first\_bucket" {

bucket = "kevholditch-bucket-outputs"

}

output "bucket\_name" {

value = aws\_s3\_bucket.first\_bucket.id

}

output "bucket\_arn" {

value = aws\_s3\_bucket.first\_bucket.arn

}

output "bucket\_information" {

value = "bucket name: ${aws\_s3\_bucket.first\_bucket.id}, bucket arn: ${aws\_s3\_bucket.first\_bucket.arn}"

}

output "all" {

value = aws\_s3\_bucket.first\_bucket

}

Project example to output an entire resource

After running the project, you will notice an output called all that has all of the attributes exported by the aws\_s3\_bucket resource. Sometimes, it can be handy to output the whole resource to the console. Normally, when you are debugging something though, you want to see the value of one of the properties.

Running the project

Clicking the terminal will run terraform init and then terraform apply. When prompted, enter yes to run the project.

## 6 Locals

### Locals in Detail

This lesson will cover Terraform local, how they work, and how to use them.

We'll cover the following

Terraform local

Project example

Project code in detail

Running the project

Locals referencing Resources

Terraform local

A local is Terraform’s representation of a normal programming language variable. Confusingly, Terraform also has a concept called a variable which is really an input (variables are covered in the variables chapter). A local can refer to a fixed value such as a string, or it can be used to refer to an expression such as two other locals concatenated together.

Project example

Let’s dive into an example:

provider "aws" {

region = "us-east-2"

}

locals {

first\_part = "hello"

second\_part = "${local.first\_part}-there"

bucket\_name = "${local.second\_part}-how-are-you-today"

}

resource "aws\_s3\_bucket" "bucket" {

bucket = local.bucket\_name

}

main.tf file of Locals project example

Project code in detail

In the code above:

We define a local block by using the keyword locals and then an opening {.

We define each local on a new line by giving it a name. The first local we define is called first\_part.

We then follow it with an = and give it a value. For the first\_part local, we give it the value of the string literal hello.

For the second local second\_part, we are use the value "${local.first\_part}-there".

As the whole expression is inside quotes, we need to use the ${ and } around our expression so Terraform evaluates it.

To reference a local, we use the expression syntax local.local\_identifier.

The second\_part local will be set to “hello-there”. In the bucket\_name local, we are using the second\_part local in the expression "${local.second\_part}-how-are-you-today" which will evaluate to hello-there-how-are-you-today.

At the bottom of the project, we define an S3 bucket and set the name to local.bucket\_name. This will create an S3 bucket with the name hello-there-how-are-you-today.

📝Note: We do not need the ${ and } as we are not inside quotes here.

We could also have set the bucket to "${local.bucket\_name}", which would be evaluated to the same thing.

In Terraform version 1.0.0, though, we can now omit the ${ and } for a single line expression where we are using the whole value. This makes the code cleaner and easier to read. Easily read code becomes vital when we define our infrastructure as code.

Locals referencing Resources

Locals can reference the output of a resource or a data source by using the expression syntax we have learned. This allows you to give something a meaningful name or to combine outputs from different resource and data source attributes to build up a more complex value.

## 7 Templates and Files

### Files

This lesson will teach you about Terraform files and how to use them.

We'll cover the following

Terraform files

Project example

Terraform files

We have seen in the locals chapter how you can place a multi-line string as a value for a property. This is useful for something like an IAM policy. It can be even cleaner to move the value out into a file and then reference that file from your project. Doing this may remove the clutter from your project and make it much more readable.

Project example

Let’s see an example of using files:

<<<main.tf>>>

provider "aws" {

region = "us-east-2"

}

resource "aws\_iam\_policy" "my\_bucket\_policy" {

name = "list-buckets-policy"

policy = file("./policy.iam")

}

<<<policy.iam>>>

{

"Version": "2012-10-17",

"Statement": [

{

"Action": [

"s3:ListBucket"

],

"Effect": "Allow",

"Resource": "\*"

}

]

}

Terraform files project example

The IAM policy creates a policy that gives list bucket permission to any bucket. If you look at the Terraform code, you will see we are configuring the AWS provider, as we will be connecting to AWS. Then we are defining an AWS IAM policy. Instead of placing the policy inline as we did in a previous chapter, we referenced the policy from the file policy.iam. We are calling the file function and passing in the argument as to where the file is.

📝Note: policy.iam is a relative file path from our current location.

### Templatefile Function

In this chapter, we will learn the templatefile function in detail.

We'll cover the following

templatefile

Project example

Code explanation

Project output

templatefile

Sometimes we want to use a file but do not know all of the values before we run the project. Other times, the values are dynamic and generated as a result of a created resource. To use dynamic values in a file, we need to use the templatefile function.

The templatefile function allows us to define placeholders in a template file and then pass their values at runtime.

Project example

Let’s dive into a simple example to see how it works. Clicking the “Run” button will run terraform init and then terraform apply. When prompted, enter yes to run the project.

<<<main.tf>>>

locals {

rendered = templatefile("./example.tpl", { name = "kevin", number = 7})

}

output "rendered\_template" {

value = local.rendered

}

<<<example.tpl>>>

hello there ${name}

there are ${number} things to say

Templatefile function example project

Code explanation

Let’s walk through the code we have written so we can understand what is going on:

We define a local (as we learned in the locals chapter) called rendered.

We set the value of the local (remember a local can have a value that is an expression) to the result of calling the templatefile function.

The template file function takes two arguments.

The first argument is the path to the template file.

In this example, we are using the relative path between the main.tf and the template file example.tpl, so the path is ./example.tpl.

The next parameter is a set of variables that we want to replace in our template.

We pass in a value for a name (kevin) and number (7).

You could set the value of these variables to any expression such as another local or an attribute from a created resource or a data source.

If you look at the example template code, you will see that we use the syntax ${<variable\_name>} when we want to reference a passed-in variable.

Terraform will replace ${name} in our template with the value passed in for the name, it will do the same with ${number}.

We then use an output to print the rendered value of the template out to the terminal. This is an easy way for us to see how the templatefile function works.

Project output

Let’s see templates in action. You will see the following output rendered:

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

Outputs:

rendered\_template =

hello there kevin

there are 7 things to say

Project example output

You can see that Terraform replaced the placeholders in our template with the values that we provided to the templatefile function. Thus, we see the message with kevin and 7 in it rather than the placeholders.

### Loops in a Template

This lesson will cover how to pass array values into a template using a loop.

We'll cover the following

Project example

Project output

Combine loops

You can pass in an array of values into a template and loop through them.

Project example

Let’s take a look at an example of how to do that:

<<<main.tf>>>

output "rendered\_template" {

value = templatefile("./backends.tpl", { port = 8080, ip\_addrs = ["10.0.0.1", "10.0.0.2"] })

}

<<<backends.tpl>>>

%{ for addr in ip\_addrs ~}

backend ${addr}:${port}

%{ endfor ~}

Project example for loops in template

This time we are just rendering the template straight to the output. Notice that we are passing in an array for ip\_addrs. Let’s look at the code in more detail:

In the template, we loop around the ip\_addrs array by using a foreach loop.

To do a foreach loop we use the syntax %{ for <var> in <variable\_name ∼} where <var> is any identifier we want to use to reference the current looped item and <variable\_name> is the name of the array that is passed into the template.

All of the lines we now write are inside the loop until we signal the end of the loop by putting %{ endfor ∼}.

Notice that inside the loop we are using the current value from the ip\_addr array and we are always referencing the port.

Project output

When we run the above project, you will notice the following output is rendered for the template:

backend 10.0.0.1:8080

backend 10.0.0.2:8080

Project example output

The word “backend” is constant as that is just text, so we see it in each iteration around the loop. The current element of the ip\_addr array is then printed, followed by the passed port, which always has the value 8080.

Combine loops

The fact that we can combine loops with interpolated values means that you can write some quite elaborate templates. These can be useful for programmatically generating things like IAM policies, where you can pass in the ARN of resources that Terraform creates and use as part of the template.

## 8 Variables

### Variables

This lesson will address Terraform variables and their uses.

We'll cover the following

Terraform variables

Our first Variable

Running the project

Destroy the project

Terraform variables

A variable in Terraform is something that can be set at runtime. It allows you to vary what Terraform will do by passing in or using a dynamic value.

Our first Variable

Let’s dive into an example of how to use variables so we can learn how they work:

provider "aws" {

region = "us-east-2"

}

variable "bucket\_name" {

description = "the name of the bucket you wish to create"

}

resource "aws\_s3\_bucket" "bucket" {

bucket = var.bucket\_name

}

Our first Terraform variable project example

You declare a variable by using the keyword variable, and then specify the identifier for the variable in quotes. We are using "bucket\_name" as the identifier. Inside the variable block, we add a description to describe to the user what this variable is used for. The description parameter is completely optional. We could have defined the variable as follows:

variable "bucket\_name" {}

📝Note: It is often a good idea to provide a description as it gives the user some indication as to what values to use and what the variable is used for.

To use the value of a variable in our Terraform code, we use the syntax var.<variable\_identifier>. We set the bucket property on the aws\_s3\_bucket resource to the value of our variable var.bucket\_name. This means that Terraform will use whatever value we give our variable as the name of the S3 bucket.

Running the project

After clicking the terminal, terraform init and terraform apply will run. Terraform will then pause and you will see the following output:

Terraform has paused because it is asking you to provide a value for the variable. The variable name is printed to the screen and underneath is the description we provided.

📝Note: If you do not set a description, only the variable name will be shown here.

Steps you should follow to run the project:

Type in a bucket name that you think will be unique. I used kevholditch-variable-bucket but it really doesn’t matter what value you use.

Press enter. You will notice that Terraform will pause again and ask if you want to apply the changes.

Type yes and create the bucket. Terraform will have created the bucket with a name of whatever value you gave it.

Destroy the project

Once you have run the project go ahead and destroy it again:

Run terraform destroy.

Terraform will ask you for a value for the variable again. It doesn’t actually matter what value you give it this time as the variable is not needed for Terraform to destroy the bucket.

Press enter and then type yes when Terraform asks you if you really want to destroy the project.

### Variable Defaults

This lesson will teach you how to set defaults in Terraform variables and their use.

We'll cover the following

Project example

Running the project

Project example

Consider the following project:

provider "aws" {

region = "us-east-2"

}

variable "bucket\_name" {

description = "the name of the bucket you wish to create"

}

variable "bucket\_suffix" {

default = "-abcd"

}

resource "aws\_s3\_bucket" "bucket" {

bucket = "${var.bucket\_name}${var.bucket\_suffix}"

}

Terraform default variables project example

We have extended the first example and added a second variable "bucket\_suffix" which has its default set to "-abcd". Setting a default on a variable means that if you do not provide a value for that variable, then the default will be used. We then use the value of the bucket\_name variable concatenated with the value of the bucket\_suffix variable for the value of the bucket name. As we use the values inside a string, we need to surround our variables with ${ and }. Otherwise, Terraform will not use the value of the variable and instead will just print the string var.bucket\_name.

Running the project

Clicking the terminal will run the terraform init command.

After running the project:

Terraform will ask you to provide a value for bucket\_name as before.

Enter a name for the bucket and press enter.

Notice that Terraform will now ask you to confirm the run by typing yes. Confirm the run and press enter.

Terraform will then create the bucket.

You may be wondering why Terraform never asked you for a value for bucket\_suffix. This is because Terraform does not need a value for bucket\_suffix since you already gave it a default value of -abcd.

The end result is that a bucket will be created with whatever name you enter for the bucket name with -abcd on the end of it.

#### GPT

在您提供的 Terraform 配置中，有两个变量：`bucket\_name` 和 `bucket\_suffix`。对于 `bucket\_suffix` 变量，由于它已经被赋予了一个默认值（`-abcd`），因此在运行 Terraform 配置时，您不必显式地提供这个变量的值。

### 如何处理变量：

1. \*\*有默认值的变量\*\*:

- 当一个变量被定义了默认值，如 `bucket\_suffix`，如果您在运行 Terraform 时没有提供一个不同的值，Terraform 将自动使用这个默认值。

- 这意味着您不需要在命令行中输入 `bucket\_suffix` 的值，除非您想覆盖默认值。

2. \*\*没有默认值的变量\*\*:

- 对于没有默认值的变量，如 `bucket\_name`，您需要在运行 Terraform 时提供值。这可以通过几种方式实现：

- 命令行参数：在运行 `terraform apply` 或 `terraform plan` 命令时使用 `-var` 参数。

- Terraform 变量文件：使用 `.tfvars` 文件或者默认的 `terraform.tfvars` 文件。

- 环境变量：设置形如 `TF\_VAR\_` 开头的环境变量。

### 示例中的应用

在您的示例中，`bucket` 资源使用这两个变量来构建 S3 存储桶的名称：

```hcl

resource "aws\_s3\_bucket" "bucket" {

bucket = "${var.bucket\_name}${var.bucket\_suffix}"

}

```

- 这里 `bucket` 的名称将是 `bucket\_name` 和 `bucket\_suffix` 的组合。

- 如果您不提供 `bucket\_name` 的值，Terraform 将会提示您输入。

- 如果您不提供 `bucket\_suffix` 的值，Terraform 将使用默认值 `-abcd`。

因此，只有当您想改变 `bucket\_suffix` 的默认值时，才需要显式地提供它的值。在大多数情况下，使用默认值是方便的，可以减少需要手动配置的变量数量。

## 9 Project Layout

## 10 Modules

### Modules Introduction

### Modules Action

#### GPT

在 AWS SQS（Amazon Simple Queue Service）中，`redrive\_policy` 是一种机制，用于自动将无法成功处理的消息转移至另一个队列，通常称为死信队列（Dead Letter Queue, DLQ）。这项策略主要用于处理和分析那些因某些原因无法被正常消费或处理的消息。

### 关键点

1. \*\*失败次数阈值\*\*:

- `redrive\_policy` 定义了一个消息可以被接收和处理的最大次数。一旦消息达到这个阈值而未被成功处理，它会被自动发送到指定的死信队列。

2. \*\*死信队列\*\*:

- 死信队列是一个普通的 SQS 队列，用于接收失败的消息。它允许开发者隔离和分析这些消息，以确定失败的原因并采取相应的措施。

3. \*\*错误处理和调试\*\*:

- 通过使用 `redrive\_policy` 和死信队列，开发者可以更好地管理消息处理中的异常情况，比如消息格式错误、处理逻辑问题或资源不足等。

4. \*\*防止消息丢失\*\*:

- 这个机制有助于确保消息不会在处理失败后丢失，而是被安全地转移到死信队列中。

### 示例

在 SQS 队列的配置中，`redrive\_policy` 可能看起来像这样：

```json

{

"maxReceiveCount": "5",

"deadLetterTargetArn": "arn:aws:sqs:region:account-id:dead-letter-queue"

}

```

这里，`maxReceiveCount` 表示消息被传送到死信队列之前允许的最大接收次数，`deadLetterTargetArn` 是死信队列的 Amazon 资源名称（ARN）。

是的，`aws\_sqs\_queue.sqs\_dead\_letter.arn` 是在 Terraform 创建 `aws\_sqs\_queue.sqs\_dead\_letter` 这个死信队列（Dead Letter Queue, DLQ）后由该资源返回的一个值。在 Terraform 中，每当您创建一个资源时，该资源会暴露一系列的属性，这些属性可以被其他资源引用。

### 属性解释

- \*\*资源类型\*\*: `aws\_sqs\_queue` 是 AWS Simple Queue Service（SQS）队列的 Terraform 资源类型。

- \*\*资源名称\*\*: `sqs\_dead\_letter` 是您在 Terraform 配置中分配给该资源的名称。

- \*\*属性\*\*: `.arn` 是该 SQS 队列资源的一个属性，表示该队列的 Amazon 资源名称（ARN）。ARN 是一个唯一标识 AWS 资源的字符串。

### 使用场景

在您的配置中，`aws\_sqs\_queue.sqs\_dead\_letter.arn` 被用作另一个 SQS 队列的红驱策略（redrive policy）的一部分。这表示：

- 当消息在主队列 (`aws\_sqs\_queue.sqs`) 中重复失败达到一定次数后，它们将被自动转移到这个死信队列。

- Terraform 会确保首先创建死信队列，并获取其 ARN，然后再创建主队列，并在创建时引用该 ARN。

这种依赖关系的自动处理是 Terraform 的一大优势，它确保了资源创建的顺序和正确性。

### Returning a Complex Type From a Module

### Module Using a Submodule

#### GPT 三方通信

这段话描述的是在 Terraform 中使用模块来简化配置 AWS 安全组规则的过程。具体来说，它涉及到在三个 AWS 安全组之间设置入站（ingress）和出站（egress）规则，以便它们可以基于指定的协议和端口进行通信。通过使用 Terraform 模块，这个过程可以更加高效和简洁。我将用中文详细解释这个过程。

### Terraform 模块化

首先，了解 Terraform 模块的概念很重要：

- \*\*模块（Modules）\*\*: 在 Terraform 中，模块是一种封装和复用配置代码的方式。一个模块可以包含一组相关的资源配置，比如安全组、EC2 实例等。

- \*\*模块化的好处\*\*: 通过模块化，可以简化复杂的配置，提高代码的重用性，使配置管理更加清晰和易于维护。

### 三方通信模块（Three-Way Cross-Talk Module）

具体到这个例子，目标是创建一个模块来管理三个 AWS 安全组之间的网络通信规则：

1. \*\*安全组（Security Groups）\*\*: 在 AWS 中，安全组充当虚拟防火墙，用于控制进出特定资源的流量。

2. \*\*入站和出站规则（Ingress and Egress）\*\*: 入站规则定义了可以进入资源的流量，而出站规则定义了可以离开资源的流量。

3. \*\*协议和端口\*\*: 这些规则将基于特定的网络协议（如 TCP、UDP）和端口号进行配置。

### 通过 Terraform 模块简化配置

在不使用 Terraform 模块的情况下，您需要为每个安全组手动设置多条规则，这在三个安全组的情况下总共需要 12 条规则（每个安全组需要对另外两个安全组设置入站和出站规则）。

- \*\*传统方法\*\*: 手动为每个安全组创建入站和出站规则，这既费时又容易出错。

- \*\*使用 Terraform 模块\*\*: 您可以创建一个模块，该模块封装了创建这些规则的逻辑。只需指定安全组和所需的协议/端口，模块就会自动创建所有必要的规则。

### 结论

通过在 Terraform 中使用一个专门的模块来设置这些安全组规则，您不仅可以大大简化配置过程，还能使代码更加整洁和易于维护。这种方法更有效率，减少了重复性劳动，并有助于减少因手动配置错误而导致的潜在问题。

#### GPT 安全组原理

在 AWS 中，安全组之间的通信实际上是指定于安全组的规则来控制不同实例（或其他类型的资源）之间的网络流量。这并不意味着安全组本身在相互通信，而是它们作为虚拟防火墙，控制着与其关联的资源之间的通信。让我们探讨一下这背后的网络架构原理：

### 安全组作为虚拟防火墙

1. \*\*资源级别的网络控制\*\*:

- 安全组在 AWS 中充当虚拟防火墙，为 EC2 实例（或绑定的其他资源类型）提供入站和出站流量控制。

- 它们是状态性的，意味着如果入站流量被允许，相应的出站响应流量也将被允许，反之亦然。

2. \*\*基于规则的流量控制\*\*:

- 安全组中的规则定义了允许进入或离开关联资源的流量类型。这些规则可以根据源/目的地址、端口和协议进行配置。

### 安全组间的规则设置

1. \*\*引用其他安全组\*\*:

- 在安全组规则中，您可以引用另一个安全组作为源或目的地。这允许来自于或者发往与该安全组关联的任何实例的流量。

- 例如，如果安全组 A 的入站规则引用了安全组 B，则所有与安全组 B 关联的实例都可以发送流量到与安全组 A 关联的实例。

2. \*\*动态和灵活的配置\*\*:

- 通过引用安全组，规则变得更加动态。无论安全组 B 关联的实例何时更改，这些更改都会自动反映在与安全组 A 的通信规则中。

### 网络架构原理

- \*\*分层安全\*\*:

- 在复杂的网络架构中，安全层次是关键。安全组为各层（如网络层、应用层）提供精细的流量控制，增强安全性。

- \*\*微分割\*\*:

- 安全组能够实现微分割，即在更细的粒度上隔离网络流量，确保只有经过严格控制和验证的流量可以在不同服务或应用之间流动。

- \*\*适应性和可扩展性\*\*:

- 随着云环境中实例的动态变化，安全组规则提供了必要的适应性和可扩展性，以响应这些变化。

总的来说，安全组之间的通信规则实际上是对实例之间流量的控制。这背后的网络架构原理是基于确保安全、灵活性和易于管理的考虑，使安全组成为控制云环境中不同资源之间交互的强有力工具。

是的，您的理解是正确的。在 AWS 中，当一个安全组（比如安全组 A）的规则引用另一个安全组（比如安全组 B）作为源（对于入站规则）或目的地（对于出站规则）时，这意味着所有与安全组 B 关联的实例都被允许与安全组 A 关联的实例进行通信，前提是这些通信符合安全组 A 的规则。

### 示例解释

- \*\*入站规则\*\*:

- 如果安全组 A 的入站规则引用了安全组 B，那么任何与安全组 B 关联的实例都可以向与安全组 A 关联的实例发送流量。这里的关键是流量必须符合安全组 A 规定的协议和端口限制。

- \*\*出站规则\*\*:

- 同理，如果安全组 A 的出站规则指定安全组 B 作为目的地，那么与安全组 A 关联的实例可以向与安全组 B 关联的实例发送流量，前提是这些通信符合出站规则的设置。

### 网络安全的灵活性

这种方法为 AWS 中的网络安全配置提供了高度的灵活性和动态性。它允许用户根据实例的安全组成员资格而不是静态的 IP 地址来定义安全规则，这在动态变化的云环境中是非常有用的。随着实例的增加或减少，或者实例在安全组间的移动，这些规则会自动适应这些变化，无需手动更新 IP 地址或其他静态标识。

如果安全组 A 的入站规则引用了安全组 B，但是安全组 A 的出站规则没有引用安全组 B，这意味着通信的单向性将受到影响。具体表现如下：

1. \*\*从安全组 B 到安全组 A 的流量\*\*:

- 安全组 A 的入站规则允许来自安全组 B 关联实例的流量。这意味着任何与安全组 B 关联的实例都可以向与安全组 A 关联的实例发送数据。

2. \*\*从安全组 A 到安全组 B 的流量\*\*:

- 由于安全组 A 的出站规则没有引用安全组 B，因此与安全组 A 关联的实例默认情况下不能向安全组 B 关联的实例发送出站流量。除非安全组 A 有其他规则允许出站流量或者安全组 B 的入站规则允许来自任何源的流量。

3. \*\*状态性行为\*\*:

- AWS 安全组是状态性的，这意味着如果一个入站请求被允许，其响应流量（即数据包的回复部分）通常会被自动允许，无论出站规则如何。这意味着即使安全组 A 的出站规则没有明确允许到安全组 B 的流量，它仍然可以发送响应流量回安全组 B，只要这些流量是对安全组 B 发起的入站通信的直接回复。

因此，在这种配置下，安全组 B 可以主动与安全组 A 通信，但安全组 A 不能主动向安全组 B 发起新的出站通信，除非有额外的出站规则允许或安全组 B 的入站规则允许。

#### GPT cross-talk-3-way

这段描述展示了如何在 Terraform 中使用模块化方法来简化 AWS 安全组间的配置。在这个例子中，使用了一个名为 `cross-talk` 的模块，它被设计用来在两个安全组之间设置跨通信（cross-talk），即相互的入站和出站规则。然后，通过在更大的模块 `cross-talk-3-way` 中引用 `cross-talk` 模块，来实现在三个安全组之间设置规则。以下是详细解读：

### `cross-talk` 模块

1. \*\*基本功能\*\*:

- `cross-talk` 模块负责在任意两个安全组之间建立通信规则。

- 它接收两个安全组、协议和端口号作为输入变量，创建相应的入站和出站规则。

### `cross-talk-3-way` 模块

1. \*\*扩展功能\*\*:

- 这个模块有一个额外的变量 `security\_group\_3`，用于接收第三个安全组资源。

- 它包含了三个 `cross-talk` 模块的实例，每个实例负责一对安全组之间的通信。

2. \*\*模块实例\*\*:

- `first\_to\_second` 实例在 `security\_group\_1` 和 `security\_group\_2` 之间建立通信规则。

- `second\_to\_third` 实例在 `security\_group\_2` 和 `security\_group\_3` 之间建立通信规则。

- `first\_to\_third` 实例在 `security\_group\_1` 和 `security\_group\_3` 之间建立通信规则。

### 整体架构

- 通过这种方式，可以在三个安全组之间有效地建立完整的通信规则。由于有三个安全组，因此需要在每一对安全组之间建立规则，总共是三对。

- 使用模块化的方法（`cross-talk` 和 `cross-talk-3-way`），Terraform 代码保持整洁和紧凑，同时也提高了代码的可读性和可维护性。

- 这是一个展示了如何通过子模块使代码更加清晰易懂的很好的例子。

总的来说，这个 Terraform 配置示例展示了如何有效地使用模块和子模块来管理复杂的安全组配置，从而简化了云环境中安全组规则的管理和维护。

## 11 Plans

## 12 State

### State Introduction

#### GPT 先有云后有tf怎么办

这段内容主要讲述了在使用Terraform进行云资源管理时，状态文件（state file）的重要性，特别是在删除资源时。以下是对每个部分的详细解读：

1. \*\*Terraform删除资源需要状态文件\*\*：Terraform是一种基础设施即代码（Infrastructure as Code）工具，用于自动化管理和配置云资源。当你想删除像AWS的子网和EC2实例这样的资源时，Terraform需要知道这些资源是由它创建的。Terraform不能猜测AWS中的哪些资源是由它的HCL（HashiCorp Configuration Language）代码创建的，因此它需要状态文件来追踪这些信息。

2. \*\*删除资源的依赖关系\*\*：在删除资源时，Terraform需要考虑资源间的依赖关系。例如，如果你有一个EC2实例依赖于一个子网，那么在删除它们时，需要先删除EC2实例，再删除子网。Terraform通过分析HCL代码中的引用关系来确定这些依赖，然后将这些信息存储在状态文件中。在删除资源时，Terraform会根据状态文件中记录的依赖顺序来正确地进行操作。

3. \*\*状态文件解决依赖问题\*\*：如果没有状态文件，Terraform需要了解所有资源类型的依赖顺序，这在实践中几乎是不可能的，因为这种依赖关系的复杂性非常高，难以维护和扩展。状态文件为解决这一问题提供了一个优雅的解决方案。它通过跟踪资源间的依赖关系，帮助Terraform在创建和删除资源时做出正确的决策。

4. \*\*资源间的引用和依赖\*\*：某些资源的值（如Amazon资源名称ARN）在创建之前可以预测，但通常更好的做法是使用资源的输出属性来引用这些值。这意味着，在编写HCL代码时，明确资源间的依赖关系非常重要。有些资源值（如VPC的ID）直到AWS创建后才能确定，因此在这些情况下，引用资源的输出属性尤为关键。

总的来说，这段内容强调了在使用Terraform管理云资源时，正确处理资源间依赖关系以及维护状态文件的重要性。

如果一个AWS基础设施是通过AWS的用户界面（UI）手动创建的，而你希望用Terraform来接管它，这个过程通常涉及以下几个步骤：

1. \*\*编写Terraform配置\*\*：首先，你需要为现有的AWS资源编写Terraform配置。这意味着你需要根据现有的AWS资源（如EC2实例、VPC等）编写对应的Terraform HCL代码。这个步骤要求你对Terraform的语法和AWS资源的配置有深入的了解。

2. \*\*使用Terraform Import\*\*：Terraform提供了一个`import`命令，允许将现有的云资源导入到Terraform的状态管理中。通过这个命令，你可以将手动创建的资源引入Terraform的状态文件。这个过程需要为每个资源提供相应的ID，并且你的Terraform配置必须与现有资源匹配。

3. \*\*验证和调整\*\*：导入资源后，你需要验证Terraform的状态文件是否准确地反映了现有的基础设施。在这个过程中，可能需要进行一些调整和优化，以确保Terraform配置与实际资源完全一致。

关于你提到的“无法厘清复杂的依赖关系”问题，这主要是指在没有任何现有配置或状态文件的情况下，Terraform无法自动识别并管理云资源间复杂的依赖关系。但通过使用Terraform的导入功能和手动编写匹配的配置，你可以逐步将现有的AWS基础设施纳入Terraform的管理范畴。

需要注意的是，这个过程可能比较繁琐且容易出错，特别是对于复杂的基础设施。因此，推荐在进行这些操作之前充分了解你的AWS资源和Terraform的相关功能。此外，在实际操作之前进行备份和测试也是非常重要的。

是的，您的理解是正确的。目前，Terraform 并不能自动生成用于描述一个 AWS 账户下所有基础设施（infrastructure）的 Terraform 配置文件（.tf 文件）。Terraform 的设计主要是面向基础设施即代码（Infrastructure as Code, IaC）的概念，即通过代码来管理和配置基础设施，而不是自动从现有的云环境生成代码。因此，如果要将已经存在的 AWS 资源纳入 Terraform 管理，需要遵循以下步骤：

1. \*\*手动编写配置\*\*：您需要为现有的 AWS 资源手动编写 Terraform 配置文件。这需要对 Terraform 的配置语言（HCL）以及 AWS 资源的结构有深入的了解。

2. \*\*使用 Terraform Import\*\*：通过 `terraform import` 命令，您可以将现有的云资源导入到 Terraform 状态中。这个过程需要手动指定要导入资源的 Terraform 资源地址和现有云资源的 ID。

3. \*\*调整和验证\*\*：导入资源后，您需要检查和调整 Terraform 配置，确保它们与实际的云资源设置相匹配。然后，您可以使用 Terraform 的计划（plan）和应用（apply）步骤来管理和更新这些资源。

由于这个过程可能比较复杂且容易出错，尤其是在处理大型或复杂的基础设施时，建议在进行这些操作前进行充分的规划和测试。

此外，市场上也存在一些第三方工具和服务，它们声称能够帮助自动生成 Terraform 配置文件，但这些工具的效果和准确性可能因具体情况而异，并且它们不是 Terraform 官方提供的。使用这些工具时需要谨慎，并且最好对生成的配置进行仔细的审核和调整。

如果您写了一个新的 Terraform 项目，其目的只是创建一个 AWS SQS（Simple Queue Service）队列，并且将其应用（`apply`）到一个已经存在的、包含多种其他资源的复杂 AWS 基础设施中，以下是可能发生的情况：

1. \*\*SQS 队列被创建\*\*：Terraform 将按照您的配置创建一个新的 SQS 队列。如果您的配置是正确的，且您有足够的权限，在 AWS 中就会出现一个新的 SQS 队列实例。

2. \*\*不会影响其他资源\*\*：如果您的 Terraform 配置只包含创建 SQS 队列的代码，且没有提及或配置其他任何资源，那么它将不会影响到 AWS 基础设施中的其他现有资源。Terraform 的操作范围限于其配置文件中定义的资源。

3. \*\*Terraform 状态更新\*\*：Terraform 将更新其状态文件，以反映新创建的 SQS 队列的状态。这个状态文件只会跟踪您用 Terraform 配置和创建的资源。

4. \*\*潜在的配置冲突\*\*：如果您的 Terraform 项目中的资源配置（如 SQS 队列的名称）与现有基础设施中的资源配置发生冲突（如已有同名的 SQS 队列），那么在执行 `apply` 操作时，您可能会遇到错误或冲突。

5. \*\*无法管理未在 Terraform 中定义的资源\*\*：Terraform 只能管理和跟踪在其配置文件中定义的资源。因此，尽管您的 Terraform 配置被应用到了一个包含多种资源的复杂基础设施中，但它不会“看到”或管理任何不在其配置文件中定义的资源。

6. \*\*最佳实践\*\*：确保您的 Terraform 配置不会与现有基础设施中的资源设置冲突，并在应用任何更改之前执行 `terraform plan` 命令来预览这些更改。

总之，只要您的 Terraform 配置专注于创建 SQS 队列且没有与现有资源产生冲突，应用该配置到现有的复杂基础设施中将只会创建您在 Terraform 中定义的 SQS 队列，而不会影响其他未在 Terraform 配置中定义的资源。

要将一个新创建的 SQS（Simple Queue Service）队列连接到基础设施中已经存在的 EC2 实例，您需要进行几个步骤，这些步骤涉及到 AWS 的配置以及可能需要更新 Terraform 代码。以下是您可以采取的步骤：

1. \*\*更新 Terraform 配置\*\*：首先，您需要在 Terraform 配置中定义新的 SQS 队列。如果您已经这样做了，并且队列已经通过 Terraform 创建，那么这一步已经完成。

2. \*\*获取 EC2 实例信息\*\*：您需要获取已经存在的 EC2 实例的相关信息。这可能包括实例的 ID、安全组、IAM 角色等。如果这个 EC2 实例不是由 Terraform 管理的，您需要手动获取这些信息。

3. \*\*配置 EC2 实例以访问 SQS\*\*：

- \*\*IAM 角色和策略\*\*：确保 EC2 实例有一个适当的 IAM 角色，该角色具有访问 SQS 队列所需的权限。您可能需要创建或更新一个 IAM 策略来包括对该 SQS 队列的访问权限，并将其附加到 EC2 实例的 IAM 角色。

- \*\*软件配置\*\*：在 EC2 实例上，您可能需要安装和配置适当的软件（如 AWS SDK、命令行工具等）来与 SQS 队列交互。

4. \*\*Terraform 导入现有资源\*\*（可选）：如果您希望将现有的 EC2 实例纳入 Terraform 的管理范围，您可以使用 `terraform import` 命令将其导入。这一步骤需要在 Terraform 配置中预先定义 EC2 实例的框架，并确保它与实际的 EC2 实例设置匹配。

5. \*\*代码和配置的一致性\*\*：如果您更改了 EC2 实例（例如，通过附加新的 IAM 角色），并且希望这些更改由 Terraform 管理，那么您需要在 Terraform 配置中反映这些更改。

6. \*\*应用和测试\*\*：应用您的 Terraform 配置更改，并在实际环境中测试 EC2 实例与 SQS 队列的连接和交互。

请注意，这个过程可能涉及到复杂的 AWS 配置，并且如果 EC2 实例最初不是通过 Terraform 创建的，将其纳入 Terraform 的管理可能是一个挑战。确保您在进行任何更改之前都有适当的备份，并在实施之前充分测试配置。

#### GPT IAM ROLE

EC2实例需要适当的IAM（Identity and Access Management）角色来访问SQS（Simple Queue Service）主要是因为在AWS中，IAM角色被用于定义和管理对AWS服务的访问权限。这些角色可以指定哪些操作可以在哪些资源上执行，从而确保安全性和最小权限原则。关于您的问题：

1. \*\*为什么需要IAM角色\*\*：当EC2实例需要与其他AWS服务（如SQS）交互时，它需要有相应的权限。这些权限通过IAM角色来赋予。例如，如果EC2实例需要向SQS发送消息，它需要拥有向该SQS队列写入消息的权限。

2. \*\*EC2实例与IAM角色\*\*：一个EC2实例在任何给定时间只能关联一个IAM角色。但是，这个角色可以包含多个权限策略，这意味着它可以被配置为允许EC2实例执行多种不同的操作和访问多种不同的AWS资源。

3. \*\*修改IAM角色\*\*：如果EC2实例已经有一个IAM角色，但这个角色不包含与SQS交互所需的权限，您可以修改这个角色的策略，增加对SQS的访问权限。这样做比替换整个角色通常更容易和更安全，特别是如果EC2实例已经有其他依赖于当前角色的权限设置。

因此，为了让EC2实例能够安全地与SQS队列交互，您需要确保它关联的IAM角色具有适当的权限策略。如果需要，这些策略可以在IAM角色中进行调整或添加。

### Manipulating State

#### GPT VPC SUBNET

在 AWS（亚马逊网络服务）中，VPC（虚拟私有云）和子网是构建云基础设施的关键组件，它们之间的关系是层级和包含性质的：

1. \*\*VPC（虚拟私有云）\*\*：VPC 是 AWS 中的一个隔离的网络环境，可以看作是您在 AWS 云中拥有的一个私有网络空间。您可以在其中定义自己的IP地址范围、创建子网、配置路由表和网络网关。VPC 提供了网络隔离和保护，确保您在 AWS 中的资源运行在一个逻辑上隔离的部分中。

2. \*\*子网\*\*：子网是 VPC 内部的一个分段。它允许您将 VPC 的 IP 地址范围划分为更小的部分，用于不同的用途（例如，公有子网和私有子网）。每个子网都位于特定的可用区中，这有助于提供高可用性和容错能力。您可以在子网中部署 AWS 资源，如 EC2 实例、RDS 数据库等。

3. \*\*关系\*\*：

- \*\*层级关系\*\*：子网是 VPC 的一部分。您不能在没有 VPC 的情况下创建子网。每个子网都必须隶属于某个 VPC。

- \*\*网络隔离\*\*：虽然所有子网都属于某个 VPC，但它们可以被配置为逻辑上相互隔离。例如，一些子网可以配置为公有（可以直接从互联网访问），而其他子网可以配置为私有（不能直接从互联网访问）。

- \*\*资源部署\*\*：您可以根据不同的需求和配置在不同的子网中部署资源。例如，您可能在公有子网中部署Web服务器，在私有子网中部署数据库服务器。

总之，在 AWS 中，VPC 提供了一个隔离和安全的网络环境，而子网则允许您在这个环境中进一步细分和控制网络结构，以适应不同的业务需求和安全要求。

理解 VPC（虚拟私有云）和子网在功能上的区别，可以从它们在网络架构中所扮演的角色和作用来入手：

1. \*\*VPC（虚拟私有云）\*\*：VPC 是 AWS 中的一项服务，它为您提供了一个隔离的虚拟网络环境。在这个环境中，您可以完全控制您的网络设置，包括IP地址范围、子网划分、路由表配置和网络网关。VPC 是在更高一级的网络层面上进行管理和隔离的。它类似于一个大型的私有网络，您可以在其中构建整个云基础设施的网络架构。

2. \*\*子网\*\*：子网是 VPC 内的一个更小的网络分段。每个子网都是 VPC IP 地址范围的一部分，它允许您对 VPC 进行更细致的网络划分。您可以在不同的子网中放置不同类型的资源（例如，将Web服务器放在公有子网中，而将数据库服务器放在私有子网中）。每个子网都位于一个特定的可用区内，这有助于实现高可用性和容错能力。

3. \*\*功能差异\*\*：

- VPC 提供了整个网络的框架和边界。它是您在 AWS 中创建的所有网络资源的基础。

- 子网允许您在 VPC 内创建逻辑上隔离的网络分段。每个子网可以有不同的安全设置和访问权限，这有助于细分网络流量和管理资源的安全性。

总的来说，VPC 是一个更广泛的概念，它代表了您在 AWS 中拥有的整个私有网络空间。而子网是在这个空间内部划分出来的小块区域，它们允许您进行更精细的网络管理和资源隔离。您可以把 VPC 看作是一个大型的建筑物，而子网则像是建筑物内部的不同房间，每个房间都有特定的用途和访问规则。

VPC（虚拟私有云）本身的 CIDR（Classless Inter-Domain Routing）块并不是 0。在 AWS 中，当您创建一个 VPC 时，您需要为它指定一个 IP 地址范围，这是通过选择一个 IPv4 CIDR 块来实现的。这个 CIDR 块定义了 VPC 中可用的 IP 地址范围。重要的是要注意以下几点：

1. \*\*CIDR 块限制\*\*：VPC 的 CIDR 块不能是任意的。它必须是私有的 IPv4 地址空间中的一部分，通常是以下三个范围之一：

- 10.0.0.0 - 10.255.255.255 (10.0.0.0/8)

- 172.16.0.0 - 172.31.255.255 (172.16.0.0/12)

- 192.168.0.0 - 192.168.255.255 (192.168.0.0/16)

这些范围是根据 RFC 1918 设计的，用于私有网络，避免与互联网上的公共 IP 地址冲突。

2. \*\*CIDR 块大小\*\*：您为 VPC 指定的 CIDR 块决定了您可以在 VPC 内分配多少 IP 地址。CIDR 块越大，可用的 IP 地址就越多。例如，一个 10.0.0.0/16 的 CIDR 块提供了 65,536 个 IP 地址，而 10.0.0.0/24 的 CIDR 块只提供了 256 个 IP 地址。

3. \*\*不是所有 IP 地址都可用\*\*：即使您指定了一个 CIDR 块，也不是块内的所有 IP 地址都可用于分配。AWS 为 VPC 内的网络服务保留了一些地址。例如，在每个子网中，第一个四个 IP 地址和最后一个 IP 地址是不可用的，因为它们被 AWS 保留。

4. \*\*VPC 内的子网\*\*：您可以在 VPC 内创建多个子网，每个子网都有自己的 CIDR 块，这必须是 VPC CIDR 块的子集。子网的 CIDR 不能重叠，并且必须完全包含在 VPC 的 CIDR 块内。

因此，虽然 VPC 提供了在其内部使用私有 IP 地址的能力，但这些地址是由您在创建 VPC 时指定的 CIDR 块限定的，并不是所有的 IPv4 地址都可用。您需要根据实际需要和网络规划来合理选择和划分 CIDR 块。

#### GPT IMPORT例子

当然，让我给您一个简单的 Terraform 导入（`terraform import`）的例子。假设您已经在 AWS 中手动创建了一个 S3 存储桶，并且现在想将其导入到 Terraform 管理中。以下是您需要遵循的步骤：

### 步骤 1：编写 Terraform 配置

首先，您需要在 Terraform 配置文件中定义这个 S3 存储桶。假设您的 S3 存储桶名称是 `my-example-bucket`，您的 Terraform 配置文件（例如 `main.tf`）可能看起来像这样：

```hcl

provider "aws" {

region = "us-west-2"

}

resource "aws\_s3\_bucket" "example" {

# 使用和实际 S3 存储桶相同的名称

bucket = "my-example-bucket"

# 其他配置...

}

```

### 步骤 2：执行 Terraform 导入

接下来，您将使用 `terraform import` 命令将已存在的 S3 存储桶导入到 Terraform 管理中。这需要指定 Terraform 中定义的资源名称和实际 AWS 资源的标识符（在这个例子中是 S3 存储桶的名称）。

在命令行中，运行以下命令：

```bash

terraform import aws\_s3\_bucket.example my-example-bucket

```

这里，`aws\_s3\_bucket.example` 是 Terraform 配置中定义的资源名称，`my-example-bucket` 是您要导入的实际 S3 存储桶的名称。

### 步骤 3：验证导入

完成导入后，您可以使用 `terraform plan` 命令来验证存储桶是否已成功导入到 Terraform 管理中。这个命令将显示 Terraform 状态和您的配置之间的任何差异。

### 注意事项

- 在执行 `terraform import` 之前，请确保您的 Terraform 配置文件中定义的资源与您想要导入的实际资源相匹配。

- `terraform import` 只修改 Terraform 状态文件，不会修改实际资源或 Terraform 配置文件。您需要确保您的配置文件准确地反映了导入资源的当前设置。

- 在导入资源之前，请确保您有适当的 AWS 凭据和权限。

通过这个过程，您可以将已存在的 AWS 资源纳入 Terraform 的管理范围，从而实现基础设施的代码化管理。

在 Terraform 中，将 `.tf` 配置文件中的资源与实际云环境中的资源关联通常是通过资源的唯一标识符完成的。这个唯一标识符的具体形式取决于资源类型。对于不同的 AWS 资源，这个标识符可能是资源的 ID、名称或其他特定属性。在我给出的 AWS S3 存储桶的例子中，使用的是存储桶的名称（`bucket`），因为在 AWS S3 中，存储桶的名称是全球唯一的，并且可以作为唯一标识符。

以下是几种常见 AWS 资源和它们在 Terraform 中的标识符：

1. \*\*S3 存储桶\*\*：使用存储桶的名称（`bucket`）作为唯一标识符。因为每个 S3 存储桶在 AWS 中都有一个唯一的名称。

2. \*\*EC2 实例\*\*：使用实例 ID（例如 `i-123abc456def7890`）作为唯一标识符。

3. \*\*VPC\*\*：使用 VPC 的 ID（例如 `vpc-1a2b3c4d`）作为唯一标识符。

4. \*\*IAM 角色\*\*：使用角色名称作为唯一标识符。

当您使用 `terraform import` 命令导入一个资源时，您需要提供 Terraform 资源类型和名称（在您的 `.tf` 文件中定义的）以及实际资源的唯一标识符。Terraform 会使用这个标识符来查找并关联实际的云资源。这就是为什么在 S3 存储桶的例子中，我使用了存储桶的名称作为导入命令的一部分。这个名称是 Terraform 用来在 AWS 中找到并关联相应存储桶的关键信息。

### Moving a Resource from One Project to Another

### Remote State

#### GPT 对状态文件进行版本控制

使用 AWS S3 加上 DynamoDB 实现 Terraform 状态的版本控制和多人协作涉及以下几个关键步骤：

### 步骤 1: 配置 S3 存储桶以存储 Terraform 状态文件

- 创建一个 AWS S3 存储桶用于存储 Terraform 状态文件。在这个存储桶上启用版本控制。版本控制允许您保存和恢复状态文件的每个版本，从而提供了一种回退机制，以防出现配置错误或其他问题。

### 步骤 2: 配置 DynamoDB 表用于状态锁定

- 创建一个 DynamoDB 表，用于 Terraform 的状态锁定。状态锁定是为了确保在一个时间点上只有一个 Terraform 操作能够对状态文件进行修改，从而防止并发操作引起的冲突。

### 步骤 3: 更新 Terraform 配置

- 在您的 Terraform 配置中，指定 S3 存储桶和 DynamoDB 表作为后端存储和状态锁定机制。例如：

```hcl

terraform {

backend "s3" {

bucket = "my-terraform-state-bucket"

key = "state/production/terraform.tfstate"

region = "us-west-2"

dynamodb\_table = "my-terraform-lock-table"

encrypt = true

}

}

```

### 示例场景：多人协作使用 Terraform 管理 AWS 资源

假设 Alice 和 Bob 是两名工程师，他们正在使用 Terraform 协作管理一个 AWS 项目。他们的 Terraform 状态文件存储在 S3 存储桶中，并使用 DynamoDB 表进行状态锁定。

#### 场景 1: 同时进行的更改尝试

- Alice 开始一个 Terraform 更改，这将自动在 DynamoDB 中创建一个状态锁。

- 同时，Bob 也尝试开始另一个更改。由于状态锁已被 Alice 的操作占用，Bob 的 Terraform 命令将等待锁释放或者显示错误信息，指出状态文件目前被锁定。

#### 场景 2: 回滚到先前状态

- 假设 Alice 应用的更改引发了一些问题，她需要回滚到之前的状态。

- Alice 可以在 S3 存储桶中找到 Terraform 状态文件的先前版本，并将其恢复为当前版本。

- 接下来，她可以使用 Terraform 命令应用这个旧版本的状态文件，将基础设施回滚到先前的状态。

### 注意事项

- 确保所有团队成员都有适当的 AWS 访问权限，能够访问 S3 存储桶和 DynamoDB 表。

- 定期检查和清理 DynamoDB 表，以防出现“孤立”的锁。

- 使用 Terraform 的版本控制策略，确保团队成员使用相同版本的 Terraform。

通过上述步骤和示例场景，您可以看到如何使用 S3 和 DynamoDB 实现 Terraform 状态的版本控制和多人协作。这种方法增加了操作的安全性和团队协作的效率。

您的理解是正确的。当您从 S3 回滚 Terraform 的状态文件（`terraform.tfstate`）到一个旧版本时，确实也需要确保 `.tf` 配置文件与该状态文件的版本相匹配。这是因为 Terraform 的状态文件和配置文件是紧密相关的，它们共同定义了您的基础设施的当前状态和期望状态。以下是相关的要点：

### 为什么需要匹配 `.tf` 配置文件和状态文件？

1. \*\*一致性\*\*：Terraform 的状态文件记录了基于 `.tf` 配置文件的最后一次成功应用的基础设施状态。如果 `.tf` 文件的内容与状态文件不一致，那么 Terraform 会试图解决这些差异，可能会导致不期望的更改。

2. \*\*避免冲突\*\*：如果状态文件被回滚到旧版本，而 `.tf` 文件仍然是较新的版本，`terraform apply` 可能会引入与旧状态文件不兼容的更改，这可能导致错误或者意外的基础设施更改。

### 如何确保版本一致性？

- \*\*版本控制系统\*\*：在使用 Git 或其他版本控制系统管理 `.tf` 文件时，确保回滚 Terraform 配置文件到与状态文件相对应的版本。

- \*\*记录和审计\*\*：保持记录 Terraform 状态文件和 `.tf` 文件更改的良好实践。这样，在需要回滚时，您可以轻松地找到匹配的版本。

### 实施回滚操作

1. \*\*回滚 `.tf` 文件\*\*：使用版本控制系统回滚 `.tf` 配置文件到与状态文件匹配的版本。

2. \*\*回滚状态文件\*\*：从 S3 恢复状态文件的旧版本。

3. \*\*验证更改\*\*：在应用任何更改之前，使用 `terraform plan` 验证预期的操作。

4. \*\*应用更改\*\*：如果 `terraform plan` 显示的更改是正确的，使用 `terraform apply` 应用更改。

### 注意

- 回滚操作应谨慎进行，特别是在生产环境中，因为这可能会对现有基础设施产生重大影响。

- 在执行回滚操作之前，建议进行充分的测试，并确保所有团队成员了解更改的范围和影响。

总之，确保 `.tf` 配置文件与 Terraform 状态文件的版本一致性是进行有效回滚操作的关键。这有助于确保 Terraform 正确地理解和操作您的基础设施，避免意外的问题。

## 13 Workspaces

### Workspaces

#### GPT

Terraform 的 workspace 是一种机制，允许您在同一套 Terraform 配置中管理多个独立的资源集。每个 workspace 可以拥有自己的 Terraform 状态文件，这意味着您可以使用相同的代码基础来管理不同的环境（如开发、测试和生产环境），而这些环境之间的资源和状态是完全隔离的。

### Workspace 的作用

1. \*\*环境隔离\*\*：您可以为不同的部署环境创建不同的 workspace，例如 `dev`、`staging` 和 `prod`。这样，您可以避免在一个环境中所做的更改影响到其他环境。

2. \*\*资源管理\*\*：在不同的 workspace 中，可以独立管理和维护资源，这有助于减少环境之间的干扰。

3. \*\*状态文件分离\*\*：每个 workspace 都有自己的状态文件，因此状态和配置信息在不同的 workspace 之间是隔离的。

### 使用 Workspace 的例子

假设您正在管理一个 AWS 基础设施，并且您有开发和生产两个环境。您可以使用 Terraform workspace 来为这两个环境分别管理资源。

#### 步骤 1: 创建 Workspace

首先，您需要为每个环境创建一个 workspace。假设您已经有了默认的 workspace，接下来您将为开发和生产环境分别创建 workspace。

```bash

terraform workspace new dev

terraform workspace new prod

```

这将创建两个新的 workspace，名为 `dev` 和 `prod`。

#### 步骤 2: 切换 Workspace

根据您当前的工作需求，您可以在这些 workspace 之间切换。例如，如果您想在开发环境中工作：

```bash

terraform workspace select dev

```

#### 步骤 3: 应用配置

当您在特定的 workspace 中时，任何 Terraform 命令都只会影响该 workspace。例如，当您在 `dev` workspace 中运行 `terraform apply` 时，只有开发环境的资源会被创建或更新。

#### 步骤 4: 管理不同的配置

您可以在 Terraform 配置中使用 `terraform.workspace` 变量来根据当前 workspace 的不同来调整配置。例如：

```hcl

resource "aws\_instance" "example" {

count = terraform.workspace == "prod" ? 5 : 1

# ... 其他配置 ...

}

```

在这个例子中，如果当前 workspace 是 `prod`，将创建 5 个实例；如果是其他 workspace（比如 `dev`），则只创建 1 个实例。

### 总结

Terraform 的 workspace 提供了一种有效的方法来在同一套配置中管理不同的环境，使得资源管理更加灵活和高效。通过为每个环境创建独立的 workspace，您可以确保环境间的隔离，同时共享相同的代码基础。

## 14 Provisioners

### Provisioners Introduction

#### GPT

Terraform 中的 provisioner 是一种工具，它允许您在创建资源之后执行脚本，这个脚本可以在远程或本地执行。HashiCorp 在 Terraform 中加入了 provisioner，主要是为了处理某些特定场景，这些场景可能无法通过您正在使用的提供者（provider）直接支持。为了完整性，这里提到了它们。然而，HashiCorp 建议仅将 provisioner 作为最后手段使用，因为它们是命令式的。这意味着 Terraform 无法像处理普通资源那样，知道如何将您对脚本所做的更改应用到实际环境中。

例如，假设您有一个 provisioner 脚本，用于在 Linux 服务器上安装一些 yum 包，然后运行一个命令。如果您想更新这个 provisioner 脚本，您需要确保新更新的脚本既适用于从未运行过该脚本的机器，也适用于已经运行过旧版本脚本的机器。如果使用 Terraform，可以避免这种情况，这也是不鼓励使用 provisioner 的主要原因之一。

总结来说，尽管 Terraform 的 provisioner 提供了执行脚本的灵活性，但它们可能会带来一些复杂性和不确定性，因为 Terraform 无法完全控制 provisioner 执行的结果。因此，在可能的情况下，最好通过 Terraform 的声明式资源管理来实现相同的功能。

#### GPT 代码解读

这段 Terraform 代码定义了一套 AWS 基础设施，包括 VPC、子网、互联网网关、路由表、安全组、SSH 密钥对和一个运行 Nginx 的 EC2 实例。下面我将逐一详细解释每个部分：

### 1. 提供者（Provider）配置

```hcl

provider "aws" {

region = "us-east-2"

}

```

这部分配置了 AWS 提供者，并指定了资源将被创建在 `us-east-2` 区域。

### 2. VPC 资源

```hcl

resource "aws\_vpc" "vpc" {

cidr\_block = "10.0.0.0/16"

}

```

这里创建了一个新的虚拟私有云（VPC），其 CIDR 块设置为 `10.0.0.0/16`，提供了较大的地址空间。

### 3. 互联网网关

```hcl

resource "aws\_internet\_gateway" "main" {

vpc\_id = aws\_vpc.vpc.id

}

```

为 VPC 创建了一个互联网网关，允许 VPC 内的资源与互联网通信。

### 4. 子网

```hcl

resource "aws\_subnet" "public" {

vpc\_id = aws\_vpc.vpc.id

cidr\_block = aws\_vpc.vpc.cidr\_block

map\_public\_ip\_on\_launch = true

availability\_zone = "us-east-1a"

}

```

在 VPC 内创建了一个公共子网。注意这里的 `availability\_zone` 设置为 `us-east-1a`，但 VPC 是在 `us-east-2` 区域创建的。这可能是一个错误，通常子网的可用区应该与 VPC 所在的区域相匹配。

### 5. 路由表和路由表关联

```hcl

resource "aws\_route\_table" "public" {

vpc\_id = aws\_vpc.vpc.id

route {

cidr\_block = "0.0.0.0/0"

gateway\_id = aws\_internet\_gateway.main.id

}

}

resource "aws\_route\_table\_association" "gateway\_route" {

subnet\_id = aws\_subnet.public.id

route\_table\_id = aws\_route\_table.public.id

}

```

创建了一个路由表，其中包含到所有目的地 (`0.0.0.0/0`) 的路由，通过之前创建的互联网网关。然后将这个路由表与公共子网关联。

### 6. 安全组

```hcl

resource "aws\_security\_group" "rules" {

// ... 略去了部分代码 ...

}

```

定义了一个安全组，包含两条入站规则：一条允许 SSH（端口 22）访问，另一条允许 HTTP（端口 80）访问。还包括一条出站规则，允许所有流量。

### 7. SSH 密钥对

```hcl

resource "aws\_key\_pair" "keypair" {

key\_name = "my\_key"

public\_key = file("nginx\_key.pub")

}

```

创建了一个 SSH 密钥对，用于 SSH 到 EC2 实例。这里假设 `nginx\_key.pub` 文件已经在 Terraform 配置文件所在的目录中。

### 8. 数据源和 EC2 实例

```hcl

data "aws\_ami" "ami" {

// ... 略去了部分代码 ...

}

resource "aws\_instance" "nginx" {

// ... 略去了部分代码 ...

}

```

这部分定义了一个数据源来查找最新的 Amazon Linux 2 AMI，并创建了一个 EC2 实例。实例配置为使用之前定义的 VPC、子网、安全组和 SSH 密钥。实例的 AMI 设置为通过数据源查找到的 AMI ID。

### 9. Provisioner

```hcl

provisioner "remote-exec" {

// ... 略去了部分代码 ...

}

```

定义了一个 `remote-exec` provisioner，用于在 EC2 实例创建后执行

这段 Terraform 代码详细定义了一个 AWS EC2 实例的创建和配置，用于运行 Nginx 服务器。我将逐部分详细解释：

### 数据源：AWS AMI

```hcl

data "aws\_ami" "ami" {

most\_recent = true

owners = ["amazon"]

filter {

name = "name"

values = ["amzn2-ami-hvm-2.0.\*-x86\_64-gp2"]

}

}

```

- 这个数据源 `aws\_ami` 用于查找 Amazon Machine Image（AMI）。

- `most\_recent = true` 确保获取最新版本的 AMI。

- `owners = ["amazon"]` 表示只搜索由 Amazon 发布的 AMI。

- 过滤条件 `filter` 用于选择特定的 AMI，这里选择的是 Amazon Linux 2 AMI。

### 资源：AWS EC2 实例

```hcl

resource "aws\_instance" "nginx" {

ami = data.aws\_ami.ami.image\_id

instance\_type = "t2.micro"

subnet\_id = aws\_subnet.public.id

vpc\_security\_group\_ids = [aws\_security\_group.rules.id]

key\_name = aws\_key\_pair.keypair.key\_name

// ... 省略 provisioner 和 connection 部分 ...

}

```

- 创建一个名为 `nginx` 的 EC2 实例。

- `ami` 使用了上面数据源查找到的 AMI ID。

- `instance\_type = "t2.micro"` 定义了实例的类型，这里是 `t2.micro`。

- `subnet\_id` 和 `vpc\_security\_group\_ids` 分别指定了实例所在的子网和使用的安全组。

- `key\_name` 指定了实例将使用的 SSH 密钥对的名称。

### Provisioner：远程执行

```hcl

provisioner "remote-exec" {

inline = [

"sudo amazon-linux-extras enable nginx1.12",

"sudo yum -y install nginx",

"sudo chmod 777 /usr/share/nginx/html/index.html",

"echo \"Hello from nginx on AWS\" > /usr/share/nginx/html/index.html",

"sudo systemctl start nginx",

]

}

```

- `remote-exec` provisioner 在 EC2 实例创建后运行。

- 使用 `inline` 脚本来安装和配置 Nginx。

- 脚本包括启用 Nginx 的软件包、安装 Nginx、更改默认网页的权限、修改默认网页内容并启动 Nginx 服务。

### 连接设置

```hcl

connection {

host = aws\_instance.nginx.public\_ip

type = "ssh"

user = "ec2-user"

private\_key = file("nginx\_key")

}

```

- 定义了如何连接到 EC2 实例来执行 `remote-exec` provisioner。

- 使用 EC2 实例的公共 IP 地址进行连接。

- 连接类型为 SSH，用户名为 `ec2-user`（Amazon Linux 的默认用户名）。

- `private\_key` 从指定的文件中读取 SSH 私钥，这里假设 `nginx\_key` 文件已在 Terraform 配置所在目录。

### 总结

这段代码在 AWS 中自动化地部署了一个配置了 Nginx 的 EC2 实例。它涵盖了从选择合适的 AMI、设置实例属性、安全组配置到通过 provisioner 自动安装和配置 Nginx 的整个过程。这是一个典型的基础设施即代码（Infrastructure as Code）的实现示例，展示了如何使用 Terraform 自动化云资源的配置和部署。

### Null Resources

## 15 Advanced Resources

### Count

### Foreach

### Lifecycle

### Depends\_on

## 16 Exercise