Manage Resources in Terraform State

developer.hashicorp.com/terraform/tutorials/certification-associate-tutorials/state-cli

Terraform stores information about your infrastructure in a state file. This state file keeps track of resources created by your configuration and maps them to real-world resources.

In this tutorial, you will create an AWS instance and security group, examine a state file, and then manipulate resources to observe how vital state is to your Terraform operations.

Terraform compares your configuration with the state file and your existing infrastructure to create plans and make changes to your infrastructure. When you run `terraform apply` or `terraform destroy` against your initialized configuration, Terraform writes metadata about your configuration to the state file and updates your infrastructure resources accordingly.

Prerequisites

This tutorial assumes that you are familiar with the usual Terraform plan/apply workflow. If you are new to Terraform, refer first to the Getting Started tutorial.

For this tutorial, you will need:

- The Terraform CLI 1.1.5+ installed locally
- An AWS account
- The AWS CLI installed
- Your AWS credentials configured locally with your access keys and a default region.

Note: This tutorial will provision resources that qualify under the AWS free-tier. If your account doesn't qualify under the AWS free-tier, we're not responsible for any charges that you may incur.

Create infrastructure and state

Clone the Learn Terraform State Management repository.

```
$ git clone https://github.com/hashicorp/learn-terraform-state.git
```

Change into the new directory.

```
$ cd learn-terraform-state
```

Review the `main.tf` file. This configuration deploys an Ubuntu EC2 instance publicly accessible on port 8080.

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = ">= 3.24.1"
    }
  }
  required_version = ">= 1.1.5"
}

variable "region" {
  description = "The AWS region your resources will be deployed"
}

provider "aws" {
  region = var.region
}

data "aws_ami" "ubuntu" {
  most_recent = true

  filter {
    name   = "name"
    values = ["ubuntu/images/hvm-ssd/ubuntu-focal-20.04-amd64-server-*"]
  }

  filter {
    name   = "virtualization-type"
    values = ["hvm"]
  }

  owners = ["099720109477"] # Canonical
}

resource "aws_instance" "example" {
  ami                    = data.aws_ami.ubuntu.id
  instance_type          = "t2.micro"
  vpc_security_group_ids = [aws_security_group.sg_8080.id]
  user_data              = <<-EOF
              #!/bin/bash
              apt-get update
              apt-get install -y apache2
              sed -i -e 's/80/8080/' /etc/apache2/ports.conf
              echo "Hello World" > /var/www/html/index.html
              systemctl restart apache2
              EOF
  tags = {
    Name = "terraform-learn-state-ec2"
  }
}

resource "aws_security_group" "sg_8080" {
  name = "terraform-learn-state-sg"
  ingress {
    from_port   = "8080"
    to_port     = "8080"
```

```
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
  // connectivity to ubuntu mirrors is required to run `apt-get update` and `apt-
get install apache2`
  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
}

output "public_ip" {
  value       = aws_instance.example.public_ip
  description = "The public IP of the web server"
}

output "public_ip" {
  value       = aws_instance.example.public_ip
  description = "The public IP of the web server"
}

output "security_group" {
  value = aws_security_group.sg_8080.id
}
```

Check for your default region with the AWS CLI. Your output should return a region in the format below.

```
$ aws configure get region
us-east-2
```

Create a new file called `terraform.tfvars` and set a `region` variable to your default AWS region.

```
region = "<YOUR-AWS-REGION>"
```

This configuration uses the AWS provider to create an EC2 instance and a security group that allows public access.

Initialize the directory.

```
$ terraform init
Initializing the backend...

Initializing provider plugins...
- Reusing previous version of hashicorp/aws from the dependency lock file
- Installing hashicorp/aws v3.26.0...
- Installed hashicorp/aws v3.26.0 (signed by HashiCorp)

Terraform has been successfully initialized!
##...
```

After Terraform initializes, apply the configuration and approve the run by typing `yes` at the prompt.

```
$ terraform apply


An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:
##...
Plan: 2 to add, 0 to change, 0 to destroy.

Changes to Outputs:
  + instance_id    = (known after apply)
  + public_ip      = (known after apply)
  + security_group = (known after apply)

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes

aws_security_group.sg_8080: Creating...
aws_security_group.sg_8080: Creation complete after 2s [id=sg-01b4904a558bcf6b2]
aws_instance.example: Creating...
aws_instance.example: Still creating... [10s elapsed]
aws_instance.example: Still creating... [20s elapsed]
aws_instance.example: Creation complete after 23s [id=i-010842178d90cd858]

Apply complete! Resources: 2 added, 0 changed, 0 destroyed.

Outputs:

instance_id = "i-010842178d90cd858"
public_ip = "3.133.127.249"
security_group = "sg-01b4904a558bcf6b2"
##...
```

Examine the state file

Now that you have applied this configuration, you have a local state file that tracks the resources Terraform created. Check your directory to confirm the `terraform.tfstate` file exists.

```
$ ls
README.md
main.tf
terraform.tfstate
new_state/
```

You should not manually change information in your state file in a real-world situation to avoid unnecessary drift between your Terraform configuration, state, and infrastructure. Any change in state could result in your infrastructure being destroyed and recreated at your next `terraform apply`.

Warning: Do not manually modify state files.

Open the `terraform.tfstate` file in your file editor.

This example contains few resources, so your actual state file is relatively small.

This file is the JSON encoded state that Terraform writes and reads at each operation. The first stanza contains information about your Terraform application.

Explore `resources` in state

The `resources` section of the state file contains the schema for any resources you create in Terraform. Review the `resources` section of this file.

```
"resources": [
 {
      "mode": "data",
      "type": "aws_ami",
      "name": "ubuntu",
      "provider": "provider[\"registry.terraform.io/hashicorp/aws\"]",
      "instances": [
        {
          "schema_version": 0,
          "attributes": {
            "architecture": "x86_64",
            "arn": "arn:aws:ec2:us-east-1::image/ami-0b287e7832eb862f8",
      ##...
    },
    ##...
]
```

The first key in this schema is the `mode`. Mode refers to the type of resource Terraform creates — either a resource (`managed`) or a data source (`data`). The `type` key refers to the resource type - in this case, the `aws_ami` type is a resource available in the `aws` provider.

```
##...
{
  "mode": "managed",
  "type": "aws_instance",
  "name": "example",
  "provider": "provider[\"registry.terraform.io/hashicorp/aws\"]",
  "instances": [
    {
      "schema_version": 1,
      "attributes": {
        "ami": "ami-0b287e7832eb862f8",
        "arn": "arn:aws:ec2:us-east-1:561656980159:instance/i-01d757a22e0685a10",
        "associate_public_ip_address": true,
        "availability_zone": "us-east-1c",
        ##...
        "public_ip": "3.18.101.45",
        ##...
        "secondary_private_ips": [],
        "security_groups": [
          "terraform-learn-state-sg"
        ],
        "source_dest_check": true,
        "subnet_id": "subnet-7ae38e36",
        "tags": {
          "Name": "terraform-learn-state-ec2"
        },
        ##...
      }
    }
  ]
},
```

The `aws_instance` type is a `managed` resource with the AMI from the `data.aws_ami` source.

The `instances` section in this resource contains the `attributes` of the resource. The `security_groups` attribute, for example, is captured in plain text in state as opposed to the variable interpolated string in the configuration file.

Terraform also marks dependencies between resources in state with the built-in dependency tree logic.

```
##...
          "dependencies": [
            "aws_security_group.sg_8080",
            "data.aws_ami.ubuntu"
          ]
##...
```

Because your state file has a record of your dependencies, enforced by you with a `depends_on` attribute or by Terraform automatically, any changes to the dependencies will force a change to the dependent resource.

Examine State with CLI

The Terraform CLI allows you to review resources in the state file without interacting with the `.tfstate` file itself. This is how you should interact with your state.

Run `terraform show` to get a human-friendly output of the resources contained in your state.

```
$ terraform show
# aws_instance.example:
resource "aws_instance" "example" {
    ami                          = "ami-0b287e7832eb862f8"
    arn                          = "arn:aws:ec2:us-east-1:561656980159:instance/i-
01d757a22e0685a10"
    associate_public_ip_address  = true
##...
}

# aws_security_group.sg_8080:
resource "aws_security_group" "sg_8080" {
    arn                    = "arn:aws:ec2:us-east-1:561656980159:security-
group/sg-0096a764b1e76f7fd"
    description            = "Managed by Terraform"
##...
}

# data.aws_ami.ubuntu:
data "aws_ami" "ubuntu" {
    architecture          = "x86_64"
    arn                   = "arn:aws:ec2:us-east-1::image/ami-0b287e7832eb862f8"
##...
}

Outputs:

instance_id = "i-05fc3562ca25ee77c"
public_ip = "3.18.101.45"
security_group = "sg-0096a764b1e76f7fd"
```

Run `terraform state list` to get the list of resource names and local identifiers in your state file. This command is useful for more complex configurations where you need to find a specific resource without parsing state with `terraform show`.

```
$ terraform state list
data.aws_ami.ubuntu
aws_instance.example
aws_security_group.sg_8080
```

Replace a resource with CLI

Terraform usually only updates your infrastructure if it does not match your configuration. You can use the `-replace` flag for `terraform plan` and `terraform apply` operations to safely recreate resources in your environment even if you have not edited the

configuration, which can be useful in cases of system malfunction. Replacing a resource is also useful in cases where a user manually changes a setting on a resource or when you need to update a provisioning script. This allows you to rebuild specific resources and avoid a full `terraform destroy` operation on your configuration. The `-replace` flag allows you to target specific resources and avoid destroying all the resources in your workspace just to fix one of them.

In older versions of Terraform, you may have used the `terraform taint` command to achieve a similar outcome. That command has now been deprecated in favor of the `-replace` flag, which allows for a simpler, less error-prone workflow. If you are using an older version of Terraform, consider upgrading or review the `taint` documentation for more information.

Tip: The `-replace` flag was introduced in Terraform 0.15.2. Ensure you are using the correct version of Terraform for this next step.

Run `terraform plan -replace="aws_instance.example"` to see the actions Terraform would take if you replaced the instance.

```
$ terraform plan -replace="aws_instance.example"

aws_security_group.sg_8080: Refreshing state... [id=sg-08a985b4f14d50fdc]
aws_instance.example: Refreshing state... [id=i-0c4f9abb21cf15fca]

Terraform used the selected providers to generate the following execution plan.
Resource actions are indicated with the following symbols:
-/+ destroy and then create replacement

Terraform will perform the following actions:

  # aws_instance.example will be replaced, as requested
-/+ resource "aws_instance" "example" {
##...
    }

Plan: 1 to add, 0 to change, 1 to destroy.

Changes to Outputs:
  ~ instance_id = "i-0c4f9abb21cf15fca" -> (known after apply)
  ~ public_ip   = "18.191.48.245" -> (known after apply)
```

As shown in the output, when you apply this change, Terraform will destroy your running instance and create a new one.

Run `terraform apply` with the `-replace` flag to force Terraform to destroy and recreate the resource.

```
$ terraform apply -replace="aws_instance.example"
aws_security_group.sg_8080: Refreshing state... [id=sg-08a985b4f14d50fdc]
aws_instance.example: Refreshing state... [id=i-0c4f9abb21cf15fca]

Terraform used the selected providers to generate the following execution plan.
Resource actions are indicated with the following symbols:
-/+ destroy and then create replacement

Terraform will perform the following actions:

  # aws_instance.example will be replaced, as requested
###...
Plan: 1 to add, 0 to change, 1 to destroy.

Changes to Outputs:
  ~ instance_id = "i-0c4f9abb21cf15fca" -> (known after apply)
  ~ public_ip   = "18.191.48.245" -> (known after apply)

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value:
```

Type `yes` when prompted to accept this update.

Using the `terraform apply` command with the `-replace` flag is the HashiCorp-recommended process for managing resources without manually editing your state file.

Move a resource to a different state file

Some of the Terraform state subcommands are useful in very specific situations. HashiCorp recommends only performing these advanced operations as the last resort.

The `terraform state mv` command moves resources from one state file to another. You can also rename resources with `mv`. The move command will update the resource in state, but not in your configuration file. Moving resources is useful when you want to combine modules or resources from other states, but do not want to destroy and recreate the infrastructure.

The `new_state` subdirectory contains a new Terraform configuration. This configuration creates a new EC2 instance named `aws_instance.example_new` and uses a data resource to use the same security group from your root configuration file. Change into the subdirectory.

```
$ cd new_state
```

Copy your `terraform.tfvars` file from your root directory.

```
cp ../terraform.tfvars .
```

Run `terraform init`.

```
$ terraform init
Initializing the backend...

Initializing provider plugins...
- terraform.io/builtin/terraform is built in to Terraform
- Reusing previous version of hashicorp/aws from the dependency lock file
- Installing hashicorp/aws v3.26.0...
- Installed hashicorp/aws v3.26.0 (signed by HashiCorp)

Terraform has been successfully initialized!
##...
```

Apply your configuration.

```
$ terraform apply
An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:
##...
Plan: 1 to add, 0 to change, 0 to destroy.

Changes to Outputs:
  + public_ip      = (known after apply)
  + security_group = "sg-01b4904a558bcf6b2"

## ...

aws_instance.example_new: Creating...
aws_instance.example_new: Still creating... [10s elapsed]
aws_instance.example_new: Still creating... [20s elapsed]
aws_instance.example_new: Creation complete after 23s [id=i-0f1adae85eba16b6f]

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

Outputs:

public_ip = "3.141.47.75"
security_group = "sg-01b4904a558bcf6b2"
```

Now, you have a second state file with a managed resource and a data source.

Move the new EC2 instance resource you just created, `aws_instance.example_new`, to the old configuration's file in the directory above your current location, as specified with the `-state-out` flag. Set the destination name to the same name, since in this case there is no resource with the same name in the target state file.

```
$ terraform state mv -state-out=../terraform.tfstate aws_instance.example_new
aws_instance.example_new
Move "aws_instance.example_new" to "aws_instance.example_new"
Successfully moved 1 object(s).
```

Note: Resource names *must* be unique to the intended state file. The `terraform state mv` command can also rename resources to make them unique.

Change into your root directory.

```
$ cd ..
```

Run `terraform state list` to confirm that the new EC2 instance, `aws_instance.example_new`, is present in the in original configuration's state file.

```
$ terraform state list
data.aws_ami.ubuntu
aws_instance.example
aws_instance.example_new
aws_security_group.sg_8080
```

Without adding the EC2 resource you moved, plan the configuration as is. Because the new EC2 instance is present in state but not in the configuration, Terraform plans to destroy the moved instance, and remove the resource from the state file.

```
$ terraform plan
aws_security_group.sg_8080: Refreshing state... [id=sg-0096a764b1e76f7fd]
aws_instance.example_new: Refreshing state... [id=i-01b9d7256cb89af93]
aws_instance.example: Refreshing state... [id=i-0db455e6d0511a954]

Terraform used the selected providers to generate the following execution plan.
Resource actions are indicated with the following symbols:
  - destroy

Terraform will perform the following actions:

  # aws_instance.example_new will be destroyed
  # (because aws_instance.example_new is not in configuration)
  - resource "aws_instance" "example_new" {
      - ami                          = "ami-0b287e7832eb862f8" -> null
      - arn                          = "arn:aws:ec2:us-east-
1:561656980159:instance/i-01b9d7256cb89af93" -> null
##...
    }

Plan: 0 to add, 0 to change, 1 to destroy.
```

Open the `main.tf` file in your root directory. Copy and paste the resource definition below.

```
resource "aws_instance" "example_new" {
  ami                    = data.aws_ami.ubuntu.id
  instance_type          = "t2.micro"
  vpc_security_group_ids = [aws_security_group.sg_8080_res_test.id]
  user_data              = <<-EOF
              #!/bin/bash
              apt-get update
              apt-get install -y apache2
              sed -i -e 's/80/8080/' /etc/apache2/ports.conf
              echo "Hello World" > /var/www/html/index.html
              systemctl restart apache2
              EOF
  tags = {
    Name = "terraform-learn-state-ec2"
  }
}
```

Apply your configuration. Your configuration now matches your state file and Terraform will not perform any changes.

```
$ terraform apply
##...
No changes. Your infrastructure matches the configuration.

Your configuration already matches the changes detected above. If you'd like to
update the Terraform state to match, create and apply a refresh-only plan:
  terraform apply -refresh-only

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

Change into your `new_state` directory.

```
$ cd new_state
```

Run `terraform destroy` and you should have no resources to destroy. Your `security_group` resource is a data source and you moved the `aws_instance` resource to another state file. Accept the changes by typing `yes` when prompted.

```
$ terraform destroy
Plan: 0 to add, 0 to change, 0 to destroy.

Changes to Outputs:
  - public_ip      = "3.141.47.75" -> null
  - security_group = "sg-01b4904a558bcf6b2" -> null

Do you really want to destroy all resources?
  Terraform will destroy all your managed infrastructure, as shown above.
  There is no undo. Only 'yes' will be accepted to confirm.

  Enter a value: yes


Destroy complete! Resources: 0 destroyed.
```

Remove a resource from state

The `terraform state rm` subcommand removes specific resources from your state file. This does not remove the resource from your configuration or destroy the infrastructure itself.

Change into your root directory.

```
$ cd ..
```

Remove your `security_group` resource from state.

```
$ terraform state rm aws_security_group.sg_8080
Removed aws_security_group.sg_8080
Successfully removed 1 resource instance(s).
```

Confirm the change by reviewing the state with `terraform state list`.

```
$ terraform state list
data.aws_ami.ubuntu
aws_instance.example
```

The removed `security_group` resource does not exist in the state, but the resource still exists in your AWS account.

Run `terraform import` to bring this security group back into your state file. Removing the security group from state did not remove the output value with its ID, so you can use it for the import.

```
$ terraform import aws_security_group.sg_8080 $(terraform output -raw
security_group)
aws_security_group.sg_8080: Importing from ID "sg-0096a764b1e76f7fd"...
aws_security_group.sg_8080: Import prepared!
  Prepared aws_security_group for import
aws_security_group.sg_8080: Refreshing state... [id=sg-0096a764b1e76f7fd]

Import successful!

The resources that were imported are shown above. These resources are now in
your Terraform state and will henceforth be managed by Terraform.
```

Note: If you were to apply the configuration before importing your resource back into state, Terraform would create a new resource, and your original infrastructure would keep running until you manually removed it.

Refresh modified infrastructure

The `terraform refresh` command updates the state file when physical resources change outside of the Terraform workflow.

Delete the original EC2 instance from your AWS account using the AWS CLI or the AWS Console. It may take a few moments for AWS to destroy your instance.

```
$ aws ec2 terminate-instances --instance-ids $(terraform output -raw instance_id)
{
    "TerminatingInstances": [
        {
            "InstanceId": "i-05fc3562ca25ee77c",
            "CurrentState": {
                "Code": 32,
                "Name": "shutting-down"
            },
            "PreviousState": {
                "Code": 16,
                "Name": "running"
            }
        }
    ]
}
```

By deleting this piece of infrastructure, you have created a difference between your state and the real-world resources mapped to it. The state file no longer reflects the reality of your environment. It may take up to five minutes for AWS to destroy your instance.

Run the `terraform refresh` command to update your state file.

```
$ terraform refresh
aws_security_group.sg_8080: Refreshing state... [id=sg-0f9a2681ee66d50e4]
aws_instance.example: Refreshing state... [id=i-048a9db19e06dae27]
##...
```

Run `terraform state list` to confirm Terraform deleted the original `aws_instance.example` resource from state.

```
$ terraform state list
data.aws_ami.ubuntu
aws_instance.example_new
aws_security_group.sg_8080
```

Tip: Your outputs still exist because Terraform stores them separately from your resources.

Your state file now reflects reality. You deleted the `aws_instance.example` and the `terraform refresh` command removed it from state.

The `terraform refresh` command *does not* update your configuration file. Run `terraform plan` to review the proposed infrastructure updates.

```
$ terraform plan
aws_security_group.sg_8080: Refreshing state... [id=sg-0f9a2681ee66d50e4]

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # aws_instance.example will be created
  ##...
 Plan: 1 to add, 0 to change, 0 to destroy.

Changes to Outputs:
  ~ public_ip = "18.221.204.118" -> (known after apply)
  ~ instance_id = "i-05fc3562ca25ee77c" -> (known after apply)
------------------------------------------------------------------------
```

Remove the original `aws_instance.example` resource, and `public_ip` and `instance_id` outputs from your `main.tf` file. This will prevent Terraform from recreating the resource in future Terraform operations.

```
##...

- resource "aws_instance" "example" {
-   ami                    = data.aws_ami.ubuntu.id
-   instance_type          = "t2.micro"
-   vpc_security_group_ids = [aws_security_group.sg_8080.id]
-   user_data              = <<-EOF
-               #!/bin/bash
-               apt-get update
-               apt-get install -y apache2
-               sed -i -e 's/80/8080/' /etc/apache2/ports.conf
-               echo "Hello World" > /var/www/html/index.html
-               systemctl restart apache2
-               EOF
-   tags = {
-     Name = "terraform-learn-state-ec2"
-   }
- }

resource "aws_security_group" "sg_8080" {
  name = "terraform-learn-state-sg"
  ingress {
    from_port   = "8080"
    to_port     = "8080"
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
  // connectivity to ubuntu mirrors is required to run `apt-get update` and `apt-
get install apache2`
  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
}

- output "instance_id" {
-   value = aws_instance.example.id
- }

- output "public_ip" {
-   value       = aws_instance.example.public_ip
-   description = "The public IP of the web server"
- }

##...
```

Apply the configuration, which will confirm that your configuration matches your state file, and remove their outputs from state. Accept the changes by typing `yes` when prompted.

```
$ terraform apply
aws_security_group.sg_8080: Refreshing state... [id=sg-01b4904a558bcf6b2]

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:

Terraform will perform the following actions:

Plan: 0 to add, 0 to change, 0 to destroy.

Changes to Outputs:
  - instance_id = "i-010842178d90cd858" -> null
  - public_ip   = "3.133.127.249" -> null

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes

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

Outputs:

security_group = "sg-01b4904a558bcf6b2"
```

Notice that Terraform changed the outputs and did not destroy any infrastructure.

Note: Terraform automatically performs a `refresh` during the `plan`, `apply`, and `destroy` operations. All of these commands will reconcile state by default, and have the potential to modify your state file.

Destroy your infrastructure

Terraform also updates your state file when you run a `terraform destroy` operation.

Destroy your infrastructure. Accept the changes by typing `yes` when prompted.

```
$ terraform destroy

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  - destroy

Terraform will perform the following actions:
##...
Plan: 0 to add, 0 to change, 1 to destroy.

Changes to Outputs:
  - security_group = "sg-01b4904a558bcf6b2" -> null

Do you really want to destroy all resources?
  Terraform will destroy all your managed infrastructure, as shown above.
  There is no undo. Only 'yes' will be accepted to confirm.


  Enter a value: yes

aws_security_group.sg_8080: Destroying... [id=sg-01b4904a558bcf6b2]
aws_security_group.sg_8080: Destruction complete after 1s

Destroy complete! Resources: 1 destroyed.
```

Your `terraform.tfstate` file still exists, but does not contain any resources. Run `terraform show` to confirm. You should not receive an output.

```
$ terraform show
```

Open the `terraform.tfstate` file in your file editor. The empty `resources` attribute confirms Terraform destroyed all your previous resources.

```
{
  "version": 4,
  "terraform_version": "1.1.5",
  "serial": 37,
  "lineage": "67d22e83-a917-29f3-80b0-b153b85a2e4a",
  "outputs": {},
  "resources": []
}
```

Next steps

In this tutorial, you created an EC2 Ubuntu instance and corresponding security group. Then, you examined your local state file and used `terraform state` to move, remove, and modify your resources across multiple configurations.

For more information about Terraform state, review the following documentation: