Store Remote State

Now you have built, changed, and destroyed infrastructure from your local machine. This is great for testing and development, but in production environments you should keep your state secure and encrypted, where your teammates can access it to collaborate on infrastructure. The best way to do this is by running Terraform in a remote environment with shared access to state.

Terraform Cloud allows teams to easily version, audit, and collaborate on infrastructure changes. It also securely stores variables, including API tokens and access keys, and provides a safe, stable environment for long-running Terraform processes.

In this tutorial, you will migrate your state to Terraform Cloud.

Prerequisites

This tutorial assumes that you have completed previous tutorials. If you have not, create a directory named `learn-terraform-aws-instance` and paste this code into a file named `main.tf`.

main.tf

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.16"
    }
  }

  required_version = ">= 1.2.0"
}

provider "aws" {
  region  = "us-west-2"
}

resource "aws_instance" "app_server" {
  ami           = "ami-08d70e59c07c61a3a"
  instance_type = "t2.micro"
}
```

Run `terraform init` to initialize your configuration directory and download the required providers. It is safe to re-run this command even if you have already done so in this directory.

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

Initializing provider plugins...
- Finding hashicorp/aws versions matching "~> 4.16"...
- Installing hashicorp/aws v4.27.0...
- Installed hashicorp/aws v4.27.0 (signed by HashiCorp)

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 has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
```

Next, apply your configuration. Type `yes` to confirm the proposed changes.

```
$ terraform apply
Terraform used the selected providers to generate the following execution plan.
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.

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: 1 added, 0 changed, 0 destroyed.
```

Terraform provisioned an AWS EC2 instance and stored data about the resource in a local state file.

Set up Terraform Cloud

If you have a HashiCorp Cloud Platform or Terraform Cloud account, log in using your existing credentials. For more detailed instructions on how to sign up for a new account and create an organization, review the Sign up for Terraform Cloud tutorial.

Next, modify `main.tf` to add a `cloud` block to your Terraform configuration, and replace `organization-name` with your organization name.

main.tf

```
terraform {
  cloud {
    organization = "organization-name"
    workspaces {
      name = "learn-tfc-aws"
    }
  }

  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.16"
    }
  }
}
```

Note: Older version of Terraform do not support the `cloud` block, so you must use **1.1.0 or higher** in order to follow this tutorial. Previous versions can use the `remote` backend block to configure the CLI workflow and migrate state.

Login to Terraform Cloud

Next, log into your Terraform Cloud account with the Terraform CLI in your terminal.

```
$ terraform login
Terraform will request an API token for app.terraform.io using your browser.

If login is successful, Terraform will store the token in plain text in
the following file for use by subsequent commands:
    /Users/<USER>/.terraform.d/credentials.tfrc.json

Do you want to proceed?
  Only 'yes' will be accepted to confirm.

  Enter a value:
```

Confirm with a `yes` and follow the workflow in the browser window that will automatically open. You will need to paste the generated API key into your Terminal when prompted. For more detail on logging in, follow the Authenticate the CLI with Terraform Cloud tutorial.

Initialize Terraform

Now that you have configured your Terraform Cloud integration, run `terraform init` to re-initialize your configuration and migrate your state file to Terraform Cloud. Enter "yes" when prompted to confirm the migration.

```
$ terraform init

Initializing Terraform Cloud...
Do you wish to proceed?
  As part of migrating to Terraform Cloud, Terraform can optionally copy your
  current workspace state to the configured Terraform Cloud workspace.

  Answer "yes" to copy the latest state snapshot to the configured
  Terraform Cloud workspace.

  Answer "no" to ignore the existing state and just activate the configured
  Terraform Cloud workspace with its existing state, if any.

  Should Terraform migrate your existing state?

  Enter a value: yes


Initializing provider plugins...
- Reusing previous version of hashicorp/aws from the dependency lock file
- Using previously-installed hashicorp/aws v4.17.0

Terraform Cloud has been successfully initialized!

You may now begin working with Terraform Cloud. Try running "terraform plan" to
see any changes that are required for your infrastructure.

If you ever set or change modules or Terraform Settings, run "terraform init"
again to reinitialize your working directory.
```

Now that Terraform has migrated the state file to Terraform Cloud, delete the local state file.

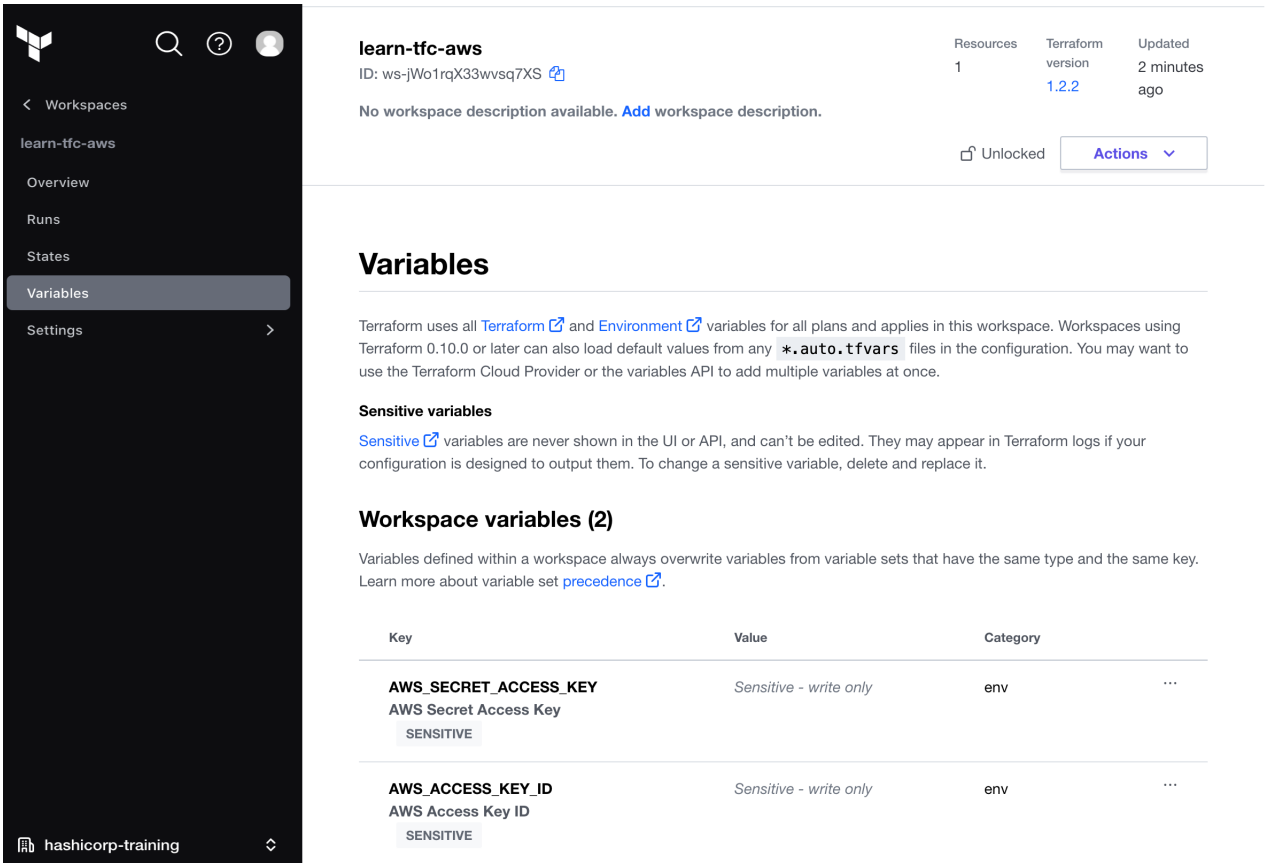
```
$ rm terraform.tfstate
```

When using Terraform Cloud with the CLI-driven workflow, you can choose to have Terraform run remotely, or on your local machine. When using local execution, Terraform Cloud will execute Terraform on your local machine and remotely store your state file in Terraform Cloud. For this tutorial, you will use the remote execution mode.

Set workspace variables

The `terraform init` step created the `learn-tfc-aws` workspace in your Terraform Cloud organization. You must configure your workspace with your AWS credentials to authenticate the AWS provider.

Navigate to your `learn-tfc-aws` workspace in Terraform Cloud and go to the workspace's **Variables** page. Under **Workspace Variables**, add your `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` as **Environment Variables**, making sure to mark them as "Sensitive".



Apply the configuration

Now, run `terraform apply` to trigger a run in Terraform Cloud. Terraform will show that there are no changes to be made.

```
$ terraform apply

## ...
No changes. Your infrastructure matches the configuration.

Terraform has compared your real infrastructure against your configuration
and found no differences, so no changes are needed.

------------------------------------------------------------------------
```

This means that Terraform did not detect any differences between your configuration and real physical resources that exist. As a result, Terraform does not need to do anything.

Terraform is now storing your state remotely in Terraform Cloud. Remote state storage makes collaboration easier and keeps state and secret information off your local disk. Remote state is loaded only in memory when it is used.

Destroy your infrastructure

Make sure to run `terraform destroy` to clean up the resources you created in these tutorials. Terraform will execute this run in Terraform Cloud and stream the output to your terminal window. When prompted, remember to confirm with a `yes`. You can also confirm

the operation by visiting your workspace in the Terraform Cloud web UI and confirming the run.

```
$ terraform destroy
Running apply in Terraform Cloud. Output will stream here. Pressing Ctrl-C
will cancel the remote apply if it's still pending. If the apply started it
will stop streaming the logs, but will not stop the apply running remotely.

Preparing the remote apply...

To view this run in a browser, visit:
https://app.terraform.io/app/hashicorp-training/learn-tfc-aws/runs/run-
kovFzCiUSrbMP3sD

Waiting for the plan to start...

Terraform v1.2.0
on linux_amd64
Initializing Terraform configuration...
aws_instance.app_server: Refreshing state... [id=i-0e756c00e19ec8f6b]

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.app_server will be destroyed
##...
Apply complete! Resources: 0 added, 0 changed, 1 destroyed.
```

Next Steps

This concludes the getting started tutorials for Terraform. Now you can use Terraform to create and manage your infrastructure.

For more hands-on experience with the Terraform configuration language, resource provisioning, or importing existing infrastructure, review the tutorials below.

- Configuration Language - Get more familiar with variables, outputs, dependencies, meta-arguments, and other language features to write more sophisticated Terraform configurations.
- Modules - Organize and re-use Terraform configuration with modules.
- Provision - Use Packer or Cloud-init to automatically provision SSH keys and a web server onto a Linux VM created by Terraform in AWS.
- Import - Import existing infrastructure into Terraform.

To read more about available configuration options, explore the Terraform documentation.

Learn more about Terraform Cloud

Although Terraform Cloud can store state to support Terraform runs on local machines, it works even better as a remote run environment. It supports two main workflows for performing Terraform runs:

- A VCS-driven workflow, in which it automatically queues plans whenever changes are committed to your configuration's VCS repo.
- An API-driven workflow, in which a CI pipeline or other automated tool can upload configurations directly.

For a hands-on introduction to the Terraform Cloud VCS-driven workflow, follow the Terraform Cloud getting started tutorials. Terraform Cloud also offers commercial solutions which include team permission management, policy enforcement, agents, and more.