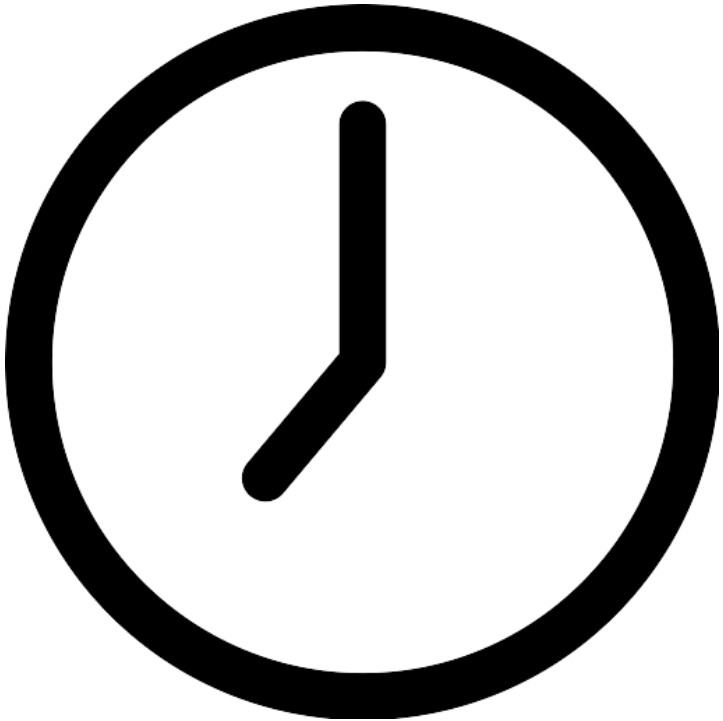
Terraform

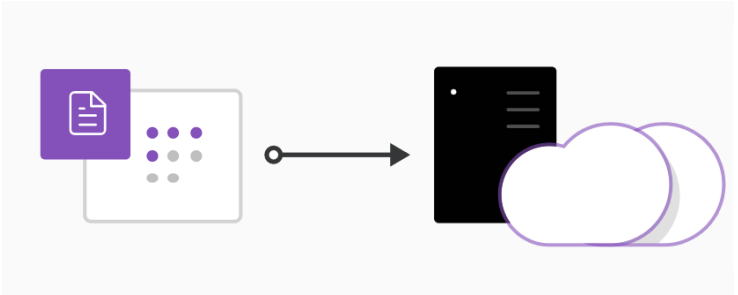
Egor Smirnov
Georgii Sergieiev
Denys Yehorov

Schedule



Infrastructure as code (IaC)

Infrastructure as code (IaC) is the process of managing and provisioning computer infrastructure using code/configuration files rather than rather than physical hardware configuration or interactive configuration tools.

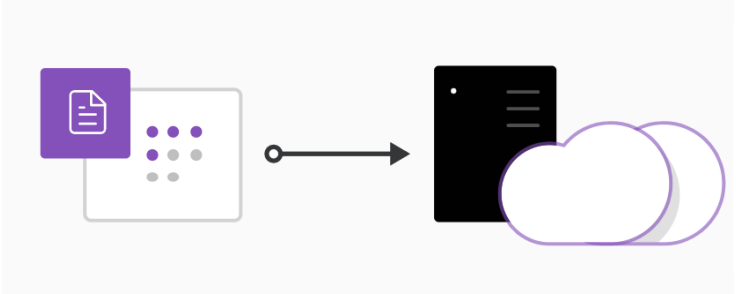


Treat infrastructure like software.

Infrastructure as code (IaC): Imperative vs. Declarative

An imperative approach defines the specific commands needed to achieve the desired configuration, and those commands then need to be executed in the correct order.

A declarative approach defines the desired state of the system, including what resources you need and any properties they should have, and an IaC tool will configure it for you.

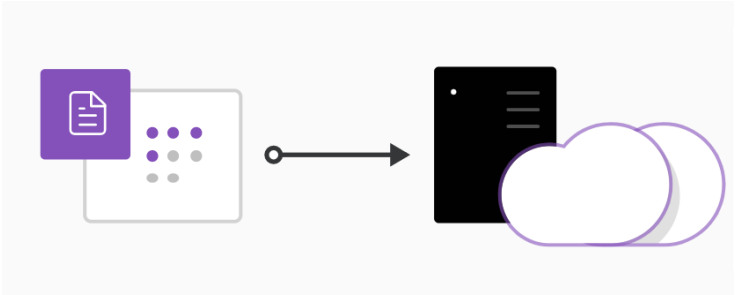


Infrastructure as code (IaC): Idempotence

Idempotence is a principle of Infrastructure as Code.

Idempotence is the property that a deployment command always sets the target environment into the same configuration, regardless of the environment's starting state.

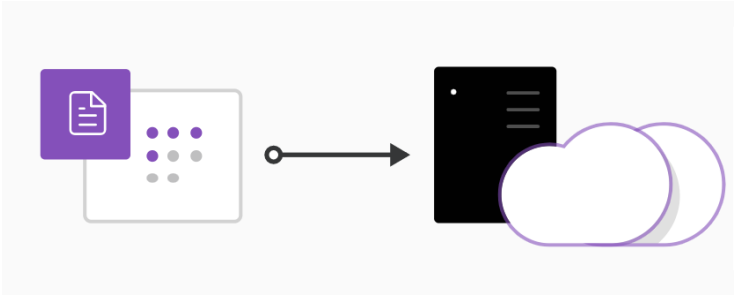
Idempotency is achieved by either automatically configuring an existing target or by discarding the existing target and recreating a fresh environment.



Infrastructure as code (IaC)

BENEFITS

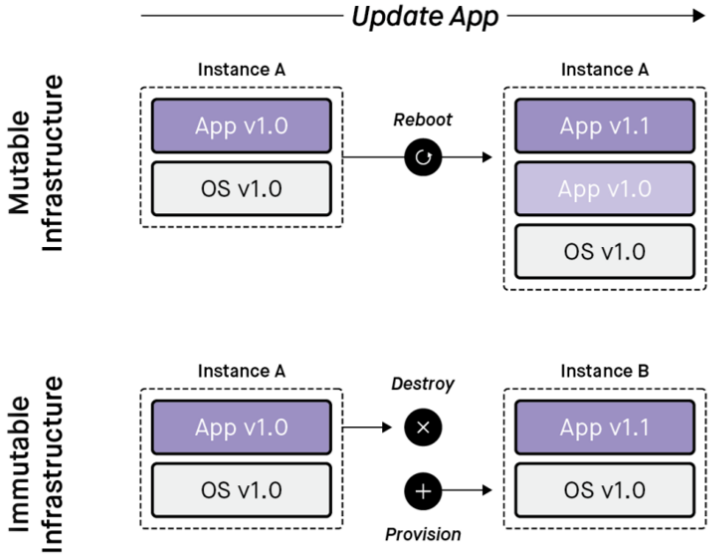
- Increase in speed of deployments
- Reduce the risk
- Write once, use anywhere
- Improve infrastructure consistency
- …



Immutable vs. mutable

Mutable infrastructure is infrastructure that can be modified or updated after it is originally provisioned

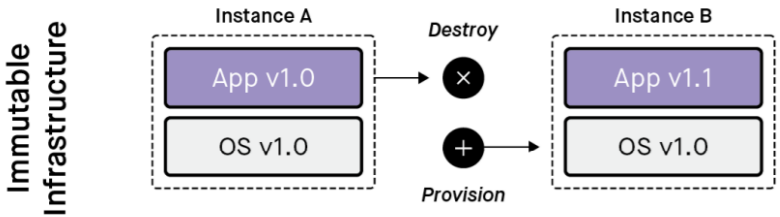
Immutable infrastructure is infrastructure that cannot be modified once originally provisioned



Immutable Infrastructure - Why ?

BENEFITS

- Predictable server state
- Predictable deployments
- Less toil work
- No configuration drift
- Simple rollback and recovery processes



What is Terraform?

HashiCorp Terraform is an infrastructure as code tool that lets you define both cloud and on-prem resources in human-readable configuration files that you can version, reuse, and share.

What is Terraform?

Initial release - 28 July 2014

Version 1.0.0 release - 08 June 2021

Latest version 1.2.3 – 15 June 2022

HashiCorp Terraform

What is Terraform?

EDITIONS

Terraform CLI

Terraform Cloud

Terraform Enterprise

What is Terraform?

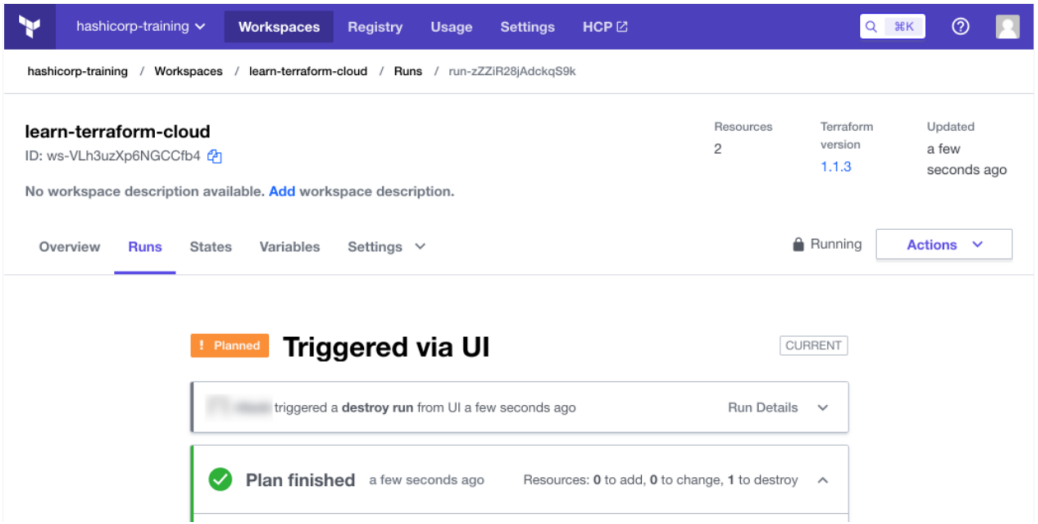
TERRAFORM CLI

```
Usage: terraform [global options] <subcommand> [args]

The available commands for execution are listed below.
The primary workflow commands are given first, followed by
less common or more advanced commands.

Main commands:
  init          Prepare your working directory for other commands
  validate      Check whether the configuration is valid
  plan          Show changes required by the current configuration
  apply         Create or update infrastructure
  destroy       Destroy previously-created infrastructure
```

HashiCorp Terraform

What is Terraform?

TERRAFORM CLOUD/TERRAFORM ENTERPRISE

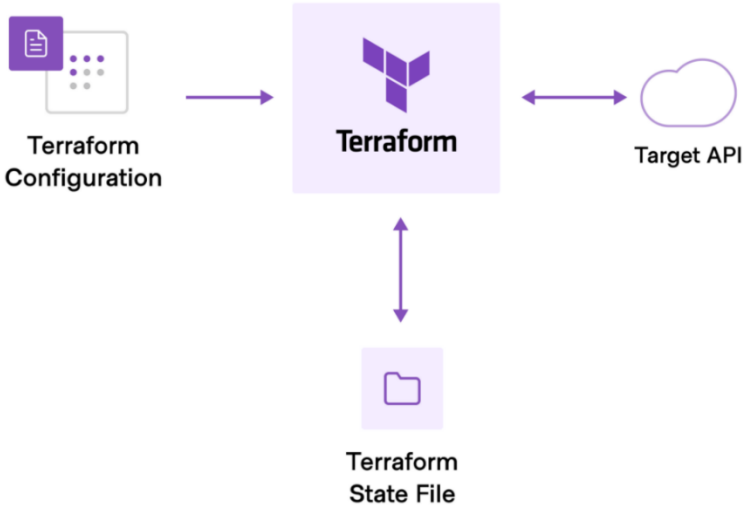


HashiCorp Terraform

How does Terraform work?

Terraform creates and manages resources on cloud platforms and other services through their application programming interfaces (APIs).

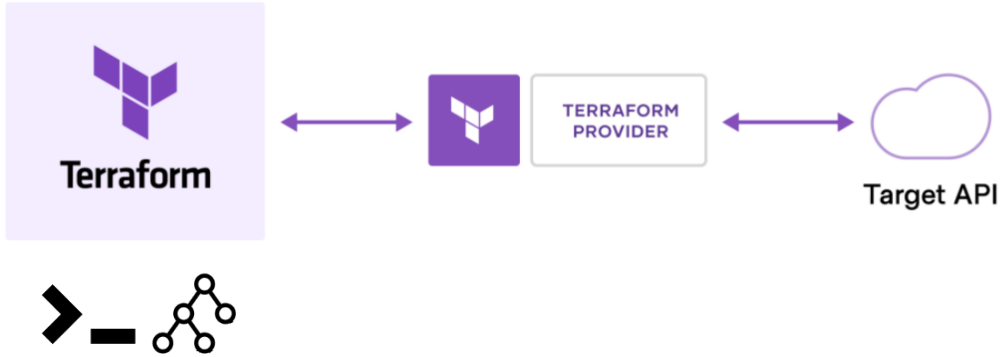
Store state about your managed infrastructure within Terraform state file.



How does Terraform work?

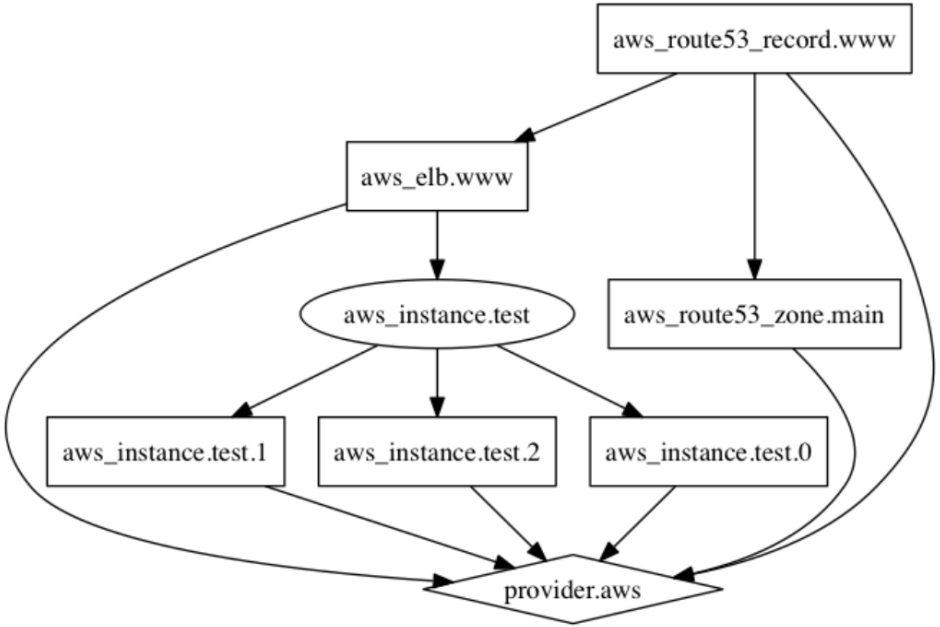
Terraform binary includes the command line interface and the main graph engine.

Providers enable Terraform to work with virtually any platform or service with an accessible API.

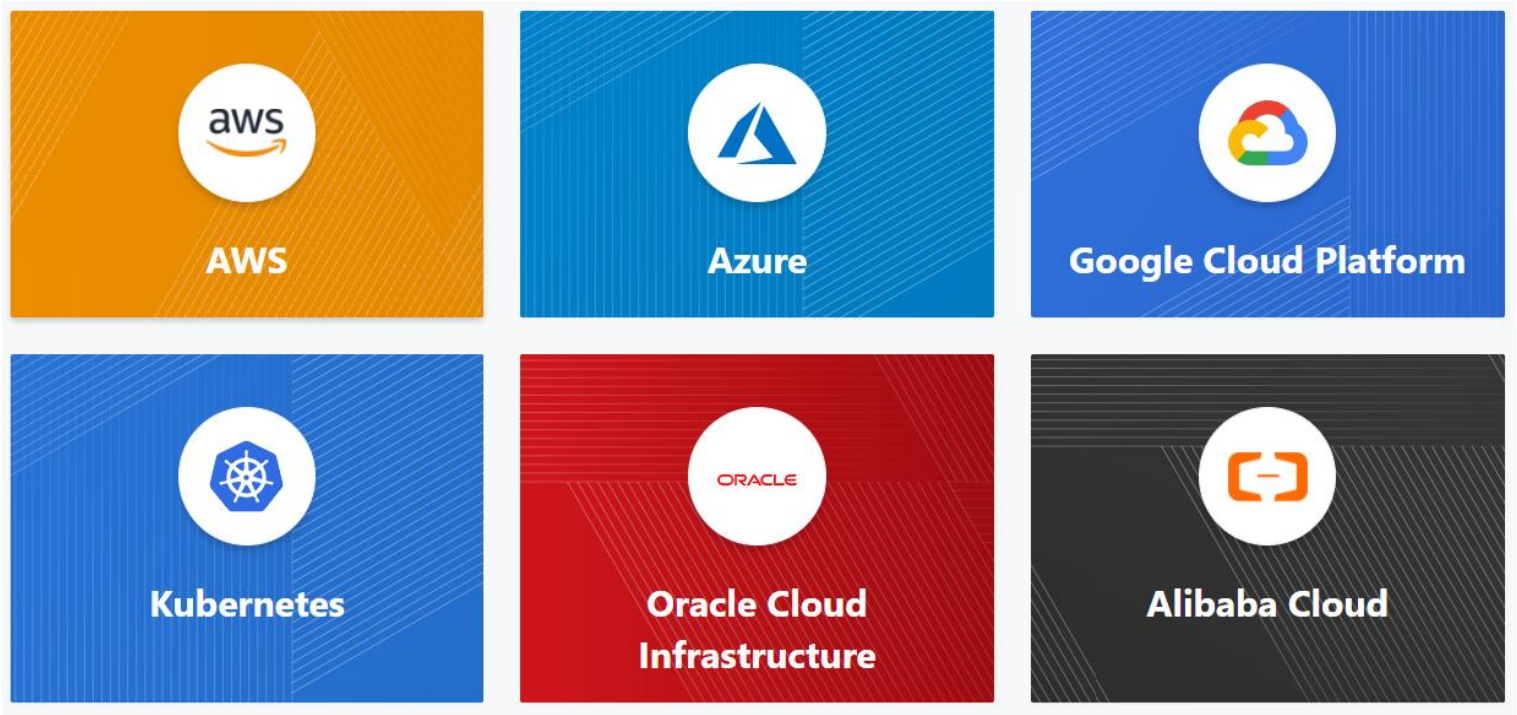


How does Terraform work?

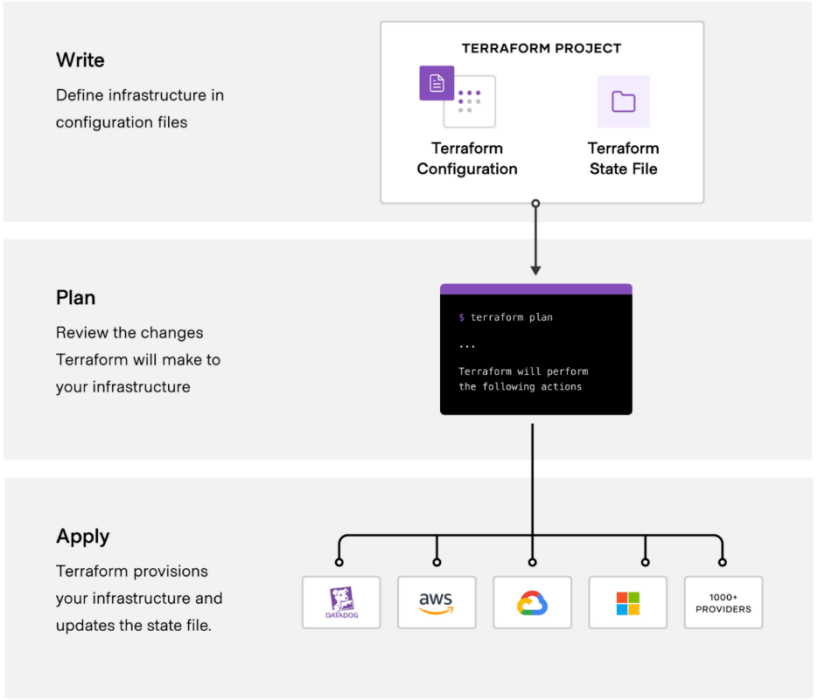
Dependency graph example



Terraform Providers



Terraform workflow



HCL

Low-level syntax of the Terraform language is defined in terms of a syntax called HCL (HashiCorp configuration language).

HCL is also used by configuration languages in other HashiCorp products (Packer, etc.).

HCL (HashiCorp configuration language) syntax is designed to be easily read and written by humans, and allows declarative logic to permit its use in more complex applications. It is attempts to strike a compromise between generic serialization formats such as YAML and configuration formats built around full programming languages such as Ruby.

The Terraform language syntax is built around two key syntax constructs: arguments and blocks.

ARGUMENTS

An argument assigns a value to a particular name:

<name> = <value>

```
image_id = "abc123"
```

BLOCKS

A block is a container for other content:

<block type> <label>…<label> {
    block body
}

```
resource "aws_instance" "example" {
  ami = "abc123"

  network_interface {
    # ...
  }
}
```

Block types

terraform { } – the special terraform configuration block type is used to configure some behaviors of Terraform itself, such as requiring a minimum Terraform version to apply your configuration.

provider { } - configure provider.

resource { } - the most important element in the Terraform language. Each resource block describes one or more infrastructure objects, such as virtual networks, compute instances, or higher-level components such as DNS records.

data { } - data sources allow Terraform to use information defined outside of Terraform, defined by another separate Terraform configuration, or modified by functions.

variable { } - input variables let you customize aspects of Terraform modules without altering the module's own source code.

output { } - output values make information about your infrastructure available on the command line, and can expose information for other Terraform configurations to use.

module { } - a module is a container for multiple resources that are used together.

…

Terraform setup

INSTALL TERRAFORM

Binary available as compiled binary from HashCorp releases page:

- https://www.terraform.io/downloads

Source code is available on github:

- https://github.com/hashicorp/terraform

VERIFY THE INSTALLATION

```
$ terraform -help
Usage: terraform [-version] [-help] <command> [args]

The available commands for execution are listed below.
The most common, useful commands are shown first, followed by
less common or more advanced commands. If you're just getting
started with Terraform, stick with the common commands. For the
other commands, please read the help and docs before usage.
##...
```

Terraform setup

AWS

Prerequisites

- Terraform
- The AWS CLI installed
- Your AWS credentials

Configure the AWS CLI

```
$ aws configure
```

AZURE

Prerequisites

- Terraform
- The Azure CLI Tool installed

Authenticate using the Azure CLI

```
$ az login
```

Standard Structure

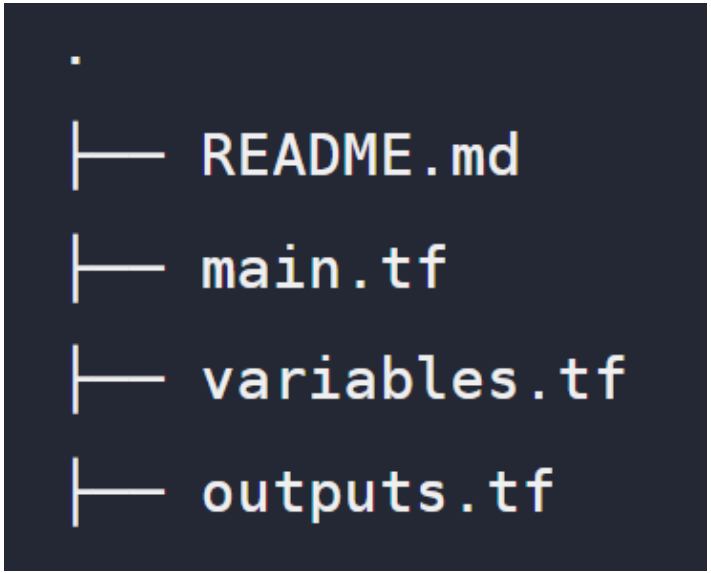
README.md - description of the code and what it should be used for.

main.tf- the primary entrypoint. For a simple cases, this may be where all the resources are created.

variables.tf - contain the declarations for variables.

outputs.tf - contain the declarations for and outputs.

There is also a JSON-based variant of the language that is named with the .tf.json file extension.



Providers

AWS

AWS Provider

https://registry.terraform.io/providers/hashicorp/aws/latest/docs

AZURE

Azure Provider

https://registry.terraform.io/providers/hashicorp/azurerm/latest/docs

Terraform init

The terraform init command is used to initialize a working directory containing Terraform configuration files.

This is the first command that should be run after writing a new Terraform configuration or cloning an existing one from version control. It is safe to run this command multiple times.

Terraform plan

The terraform plan command creates an execution plan, which lets you preview the changes that Terraform plans to make to your infrastructure.

The plan command alone will not actually carry out the proposed changes, and so you can use this command to check whether the proposed changes match what you expected before you apply the changes or share your changes with your team for broader review.

Terraform apply

The terraform apply command performs a plan just like terraform plan does, but then actually carries out the planned changes to each resource using the relevant infrastructure provider's API.

By default, terraform apply performs a fresh plan right before applying changes, and displays the plan to the user when asking for confirmation.

Terraform destroy

The terraform destroy command is a convenient way to destroy all remote objects managed by a particular Terraform configuration.

This command is just a convenience alias for the following command:

terraform apply -destroy

Terraform State file

This state is used by Terraform to map real world resources to your configuration, keep track of metadata, and to improve performance for large infrastructures.

State is stored in a local file named terraform.tfstate, but it can also be stored remotely.

Terraform state can contain sensitive data, depending on the resources in use and your definition of "sensitive." The state contains resource IDs and all resource attributes. For resources such as databases, this may contain initial passwords.

When using local state, state is stored in plain-text JSON files.

Add terraform.tfstate file into .gitignore file!

Terraform Remote State

With remote state, Terraform writes the state data to a remote data store, which can then be shared between all members of a team.

Remote state is implemented by a backend which you can configure in your configuration.

Terraform supports storing state in:

- Terraform Cloud
- HashiCorp Consul
- Amazon S3
- Azure Blob Storage
- Google Cloud Storage
- and more.

Terraform Remote State

Remote backend	Supports state locking
Azure Blob Storage	Yes
S3	Yes (supports state locking and consistency checking via Dynamo DB)
Google Cloud Storage	Yes
Consul	Yes
Alibaba Cloud OSS	Yes (supports state locking and consistency checking via Alibaba Cloud Table Store)
Artifactory (Generic HTTP repositories)	No
…	…

Terraform input variables

Input variables let you customize aspects of Terraform modules without altering the module's own source code. This allows you to share modules across different Terraform configurations, making your module composable and reusable.

Each input variable accepted by a module must be declared using a variable block.

Terraform input variables

Optional arguments for variable declarations:

- default - A default value which then makes the variable optional.
- type - This argument specifies what value types are accepted for the variable.
- description - This specifies the input variable's documentation.
- validation - A block to define validation rules, usually in addition to type constraints.
- sensitive - Limits Terraform UI output when the variable is used in configuration.
- nullable - Specify if the variable can be null within the module.

Terraform input variables

Other ways to set variables values:

- Environment variables. TF_VAR_variable_name=“value”
- –var option. Terraform apply –var=“variable_name=value”
- -var-file option. Terraform apply –var-file vars.tfvars
- Define variable value in terraform.tfvars file or *.auto.tfvars files

Secrets

Do not hard-code secrets!

Define variable for secret in variables.tf file without default value.

Use one of the following approaches to provide secret to terraform:

- Via Environment variables. TF_VAR_secret_variable_name="value"
- Via –var option. Terraform apply –var="secret_variable_name=value"
- Define variable value in terraform.tfvars file. Add terraform.tfvars into .gitignore file
- Use secret storage, like HC Vault, AWS Secret Manager, AZURE Key Vault, GCP Secret Manager, etc.

Terraform output variables

Output values make information about your infrastructure available on the command line, and can expose information for other Terraform configurations to use. Output values are similar to return values in programming languages.

Terraform modules

Modules are containers for multiple resources that are used together. A module consists of a collection of .tf and/or .tf.json files kept together in a directory.

Modules are the main way to package and reuse resource configurations with Terraform.

```
$ tree complete-module/
.
├── README.md
├── main.tf
├── variables.tf
├── outputs.tf
├── ...
├── modules/
│   ├── nestedA/
│   │   ├── README.md
│   │   ├── variables.tf
│   │   ├── main.tf
│   │   ├── outputs.tf
│   ├── nestedB/
│   ├── .../
```

Terraform modules source

```
module "consul" {
  source = "./consul"
}
```

The module installer supports installation from a number of different source types, as listed below.

- Local paths
- Terraform Registry
- GitHub
- Bitbucket
- Generic Git, Mercurial repositories
- HTTP URLs
- S3 buckets
- GCS buckets
- Modules in Package Sub-directories

THANK YOU