

Terraform: Infrastructure as code

Aleksandr Usov

Senior System Engineer

HashiCorp Certified: Terraform Associate

Agenda

- Infrastructure as code
- Terraform configuration
- Terraform settings

Evolution of Management

- Manual(documentation?)
- Scripts
- Infrastructure as code
- Immutable Infrastructure

Infrastructure as code

Treat infrastructure like software. Provisioning infrastructure through machine-readable definition files, rather than physical hardware configuration or interactive configuration tools.

Added value and advantages:

- cost (reduction)
- speed (faster execution)
- risk (remove errors and security violations)

Infrastructure as code

- version control system
- testing
- pull requests
- DRY

Types of approaches

- **declarative** (functional): you specify the desired final state of the infrastructure you want to provision and the IaC software handles the rest
- **imperative** (procedural): helps you prepare automation scripts that provision your infrastructure

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

Methods

Two methods of IaC:

- **push**: the server to be configured will pull its configuration from the controlling server
- **pull**: the controlling server pushes the configuration to the destination system

Tools

Configuration management: Ansible, Chef, Puppet, SaltStack	Infrastructure provisioning: Terraform, CloudFormation, Heat
OS Configuration	Infrastructure Automation
Application Installation	VM and Cloud Provisioning
Declarative	Declarative
Limited Infrastructure Automation	Limited OS Configuration Management

Comparison: Stack Exchange

Tool	Result	Tag
Terraform	14,733	4971
CloudFormation	9,547	4557
Azure Resource Templates	1801	1806
Google Cloud Deployment Manager	250	174

Comparison: Stack Exchange

Tool	Jobs
Terraform	64
Ansible	53
CloudFormation	19
Puppet	17
SaltStack	4

Terraform

Terraform is a tool for building, changing, and versioning infrastructure safely and efficiently. Terraform can manage existing and popular service providers.

The key features of Terraform are:

Infrastructure as Code: Infrastructure is described using a high-level configuration syntax.

Execution Plans: Terraform has a "planning" step where it generates an execution plan. The execution plan shows what Terraform will do when you call apply.

Terraform

Resource Graph: Terraform builds a graph of all your resources, and parallelizes the creation and modification of any non-dependent resources.

Change Automation: Complex changesets can be applied to your infrastructure with minimal human interaction.

Terraform

Open Source	Terraform Cloud	Terraform Enterprise
Infrastructure as code provisioning and management for any infrastructure	Collaboration and automation for practitioners and small teams using Terraform	Private installation of Terraform with collaboration, policy & governance, and self-service infrastructure for organizations

HCL

- HCL is not a format for serializing data structures(like JSON, YAML, etc). HCL is a syntax and API for building structured configuration formats
- HCL attempts to strike a compromise between generic serialization formats such as YAML and configuration formats built around full programming languages such as Ruby

go

- Knowledge of the Go language is not required, but it's better to be able to read provider code
- Provider code is a very subtle layer for cloud or service API
- Providers themselves are executable files that communicate with TF via gRPC

Installation: binary

```
$ wget 'https://releases.hashicorp.com/terraform/0.12.24/terraform_0.12.24_linux_amd64.zip'
$ unzip terraform_0.12.24_linux_amd64.zip
$ strings terraform | grep goenv | tail -1
/opt/goenv/versions/1.12.13/src/internal/cpu/cpu.go
$ strings terraform | grep teamcity | tail -1
/opt/teamcity-agent/work/9e329aa031982669/pkg/mod/github.com/...
$ ./terraform version
Terraform v0.12.24
```

Version 0.12

- May 22, 2019 → March 19, 2020
- <https://github.com/hashicorp/terraform/blob/v0.12/CHANGELOG.md>
- Current version

Code Organization

The Terraform language uses configuration files that are named with the `.tf` file extension. There is also a JSON-based variant of the language that is named with the `.tf.json` file extension.

A module is a collection of `.tf` or `.tf.json` files kept together in a directory. The root module is built from the configuration files in the current working directory when Terraform is run, and this module may reference child modules in other directories, which can in turn reference other modules, etc.

The simplest Terraform configuration is a single root module containing only a single `.tf` file.

Arguments, Blocks, and Expressions

```
<BLOCK TYPE> "<BLOCK LABEL>" "<BLOCK LABEL>" {  
  # Block body  
  <IDENTIFIER> = <EXPRESSION> # Argument  
}
```

- **Blocks** are containers for other content and usually represent the configuration of some kind of object, like a resource
- **Arguments** assign a value to a name. They appear within blocks
- **Expressions** represent a value, either literally or by referencing and combining other values

Provider Configuration

A provider configuration is created using a provider block:

```
provider "aws" {  
  version = "~> 2.0"  
  region  = "us-east-1"  
}
```

The name given in the block header ("aws" in this example) is the name of the provider to configure.

The body of the block (between { and }) contains configuration arguments for the provider itself.

terraform init

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

Usage: `terraform init [options] [DIR]`

- Copy a Source Module
- Backend Initialization
- Child Module Installation
- Plugin Installation(~/.terraform.d/plugins)

terraform init

```
Initializing the backend...
```

```
Initializing provider plugins...
```

- Checking for available provider plugins...
- Downloading plugin for provider "aws" (hashicorp/aws) 2.59.0...

The following providers **do not** have **any version constraints in** configuration, so the latest **version** was installed.

To prevent **automatic** upgrades **to new major versions** that may contain breaking changes, it **is** recommended to **add version = "..."** constraints to the **corresponding** provider blocks **in** configuration, **with** the **constraint** strings suggested below.

```
* provider.aws: version = "~> 2.59"
```

Resource Syntax

```
resource "aws_instance" "web" {  
  ami          = "ami-a1b2c3d4"  
  instance_type = "t2.micro"  
}
```

A resource block declares a resource of a given type ("aws_instance") with a given local name ("web"). Each resource type in turn belongs to a provider, which is a plugin for Terraform that offers a collection of resource types.

terraform apply

The terraform apply command is used to apply the changes required to reach the desired state of the configuration

Usage: `terraform apply [options] [dir-or-plan]`

- `-state=path` - path to the state file
- `-var 'foo=bar'` - set a variable
- `-var-file=foo` - set variables from a variable file

State

Terraform uses this local state to create plans and make changes to your infrastructure. Prior to `plan` and `apply` operation, Terraform does a refresh to update the state with the real infrastructure. The `terraform refresh` command is used to reconcile the state Terraform knows about (via its state file) with the real-world infrastructure.

State is stored by default in a local file named `terraform.tfstate`.

The state is in JSON format.

If you make a mistake modifying your state, the state CLI will always have a backup available for you that you can restore.

terraform.tfstate

```
{
  "version": 4,
  "terraform_version": "0.12.24",
  "serial": 1,
  "lineage": "6ef09c9d-f7ec-2f35-3fd0-af5fe8e3b53a",
  "outputs": {
    "all_server_ips": {
      "value": "value",
      "type": "string"
    }
  },
  "resources": []
}
```

aws_s3_bucket

```
resource "aws_s3_bucket" "main" {  
  bucket = "cf08973879519fa5610bc8b6ff6541"  
}
```

aws_s3_bucket: 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:

```
# aws_s3_bucket.main will be created
+ resource "aws_s3_bucket" "main" {
  + acceleration_status = (known after apply)
  + acl                 = "private"
  + arn                 = (known after apply)
  + bucket              = "cf08973879519fa5610bc8b6ff6541"
  + bucket_domain_name = (known after apply)
  + bucket_regional_domain_name = (known after apply)
  + force_destroy       = false
  + hosted_zone_id      = (known after apply)
  + id                  = (known after apply)
  + region              = (known after apply)
  + request_payer       = (known after apply)
  + website_domain      = (known after apply)
  + website_endpoint    = (known after apply)

  + versioning {
    + enabled    = (known after apply)
    + mfa_delete = (known after apply)
  }
}
```

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

terraform destroy

The terraform destroy command is used to destroy the Terraform-managed infrastructure.

Usage: `terraform destroy [options] [dir]`

terraform destroy

```
- versioning {  
  - enabled      = false -> null  
  - mfa_delete = false -> null  
}  
}
```

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

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_s3_bucket.main: Destroying... [id=cf08973879519fa5610bc8b6ff6541]

aws_s3_bucket.main: Destruction complete after 1s

Input Variables

Input variables serve as parameters for a Terraform module, allowing aspects of the module to be customized without altering the module's own source code, and allowing modules to be shared between different configurations

```
variable "availability_zone_names" {  
    type      = list(string)  
    default = ["us-west-1a"]  
}
```

Invalid names for variables: `source`, `version`, `providers`, `count`, `for_each`, `lifecycle`, `depends_on`, `locals`

Type Constraints

Primitive Types:

- `string` : such as "Epam"
- `number` : such as 6.283
- `bool` : either true or false

Complex Types:

- Collection Types
- Structural Types

Complex Types

Collection Types:

- `list(...)` : a sequence of values identified by consecutive whole numbers starting with zero
- `map(...)` : a collection of values where each is identified by a string label
- `set(...)` : a collection of unique values that do not have any secondary identifiers or ordering

Structural Types:

- `object(...)` : a collection of named attributes that each have their own type.
Schema: `{ <KEY> = <TYPE>, <KEY> = <TYPE>, ... }`
- `tuple(...)` : a sequence of elements identified by consecutive whole numbers starting with zero, where each element has its own type. Schema: `[<TYPE>, <TYPE>, ...]`

Root Module Variables

Root module variables can be set in a number of ways:

- Individually, with the `-var` command line option.
- In variable definitions (`.tfvars`) files, either specified on the command line or automatically loaded.
- As environment variables.
- From console input

Variable Definitions (.tfvars) Files

To set lots of variables, it is more convenient to specify their values in a variable definitions file:

```
terraform apply -var-file="prod.tfvars"
```

A variable definitions file uses the same basic syntax as Terraform language files:

```
image_id = "ami-abc123"
```

Terraform also automatically loads a number of variable definitions files if they are present:

- Files named exactly terraform.tfvars or terraform.tfvars.json.
- Any files with names ending in .auto.tfvars or .auto.tfvars.json.

terraform show

The terraform show command is used to provide human-readable output from a state or plan file

Usage: `terraform show [options] [path]`

```
terraform show
# aws_s3_bucket.main:
resource "aws_s3_bucket" "main" {
  acl                = "private"
  arn                = "arn:aws:s3:::cf08973879519fa5610bc8b6ff6541"
  bucket             = "cf08973879519fa5610bc8b6ff6541"
  bucket_domain_name = "cf08973879519fa5610bc8b6ff6541.s3.amazonaws.com"
  bucket_regional_domain_name = "cf08973879519fa5610bc8b6ff6541.s3.us-east-2.amazonaws.com"
  force_destroy      = false
  hosted_zone_id     = "Z201EMR09K5GLX"
  id                 = "cf08973879519fa5610bc8b6ff6541"
  region             = "us-east-2"
  request_payer      = "BucketOwner"

  versioning {
    enabled      = false
    mfa_delete = false
  }
}
```

terraform output

The terraform output command is used to extract the value of an output variable from the state file.

Usage: `terraform output [options] [NAME]`

```
output "arn" {  
    value = aws_s3_bucket.main.arn  
}
```

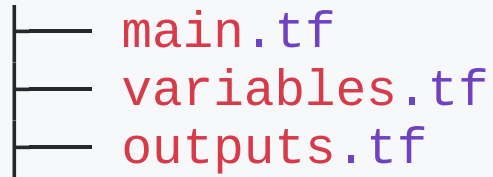
```
$ terraform output  
arn = arn:aws:s3:::cf08973879519fa5610bc8b6ff6541
```

Built-in Functions

The Terraform language includes a number of built-in functions that you can call from within expressions to transform and combine values

```
> max(-1, 4)
4
> file("/etc/debian_version")
buster/sid
> merge({"a"="1", "b"="2"}, {"c"="3", "d"="4"})
{
  "a" = "1"
  "b" = "2"
  "c" = "3"
  "d" = "4"
}
> toset(["a", "b", "a"])
[
  "a",
  "b",
]
```

Simple Structure



```
├── main.tf
├── variables.tf
└── outputs.tf
```

- `main.tf` : contains the main set of configuration
- `variables.tf` : contains the variable definitions
- `outputs.tf` : contains the output definitions

don't commit:

- `terraform.tfstate` and `terraform.tfstate.backup` : contain your Terraform state
- `.terraform` : contains the modules and plugins

Links

[Terraform Documentation](#)

End