TERRAFORM LESSON

Francisco Javier Moreno Diaz
fmorenod@gmail.com

Agenda

- IaC
- Terraform
- Lifecycle (and its commands)
- Blocks: Providers, Resource, Data, etc.
- Setup
- Exercise

Terms

Template or Source Code)

State file

Stack

What is IaC ?

Infrastructure as Code (IaC) is the process of managing and provisioning ~~cloud~~ infrastructure with ~~machine-readable~~ definition files.

Think of it as executable documentation.

Provide a codified workflow to create infrastructure (including changes/updates – version and scale)

Integrate with application code workflows (Git, CI/CD tools)

Provide reusable (repetable & auditable) modules for easy sharing and collaboration

Safely test changes using native-tools of IaC or 3rd party

Enforce security policy and organizational standards

To sum all, it offers speed, reliability, collaboration, rollback

What is Terraform?

HashiCorp
an IBM Company

HashiCorp
Terraform

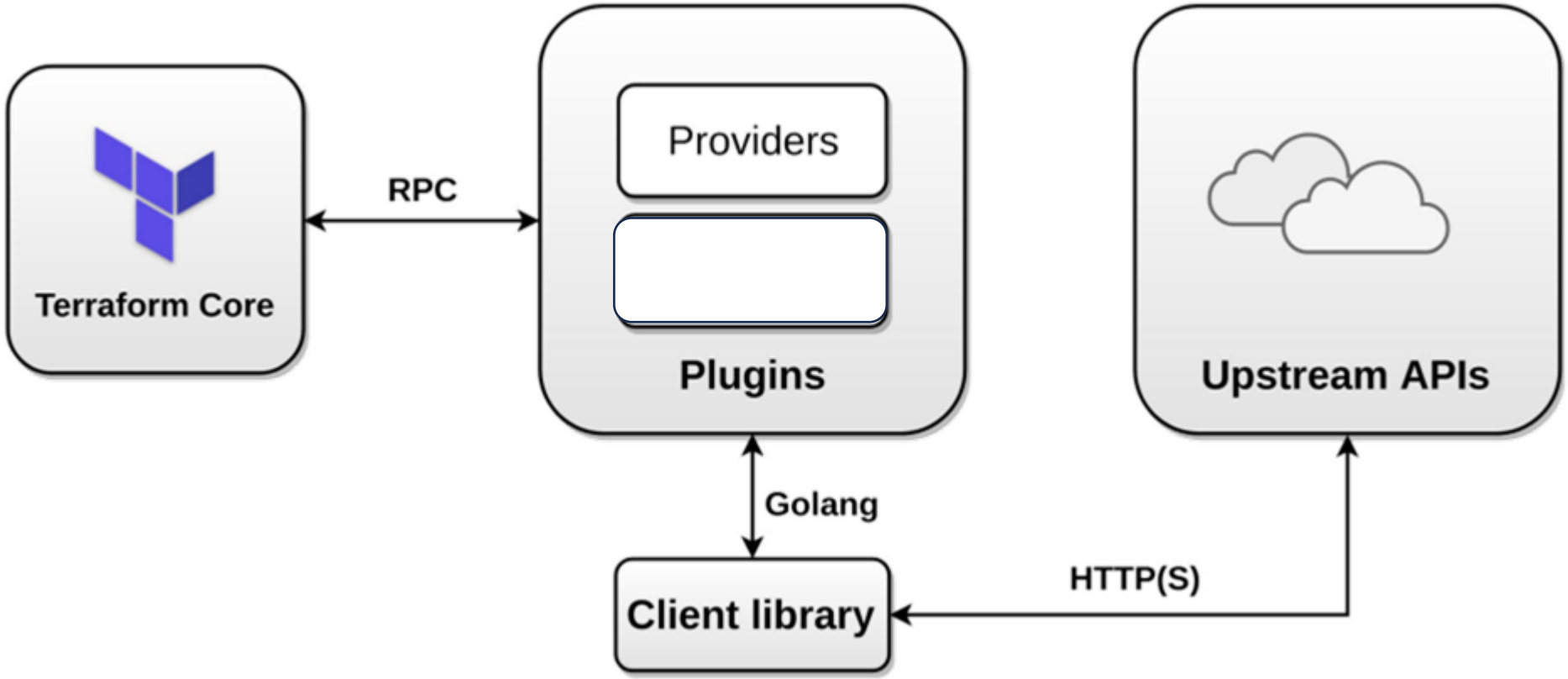
Declarative Open-source provisioning tool (IaC).

It ships as a cross-platform single binary which is written in Go.

Executable Documentation: Human and machine readable
Easy to learn
Test, share, re-use, automate
Works on all major cloud providers

OpenTofu

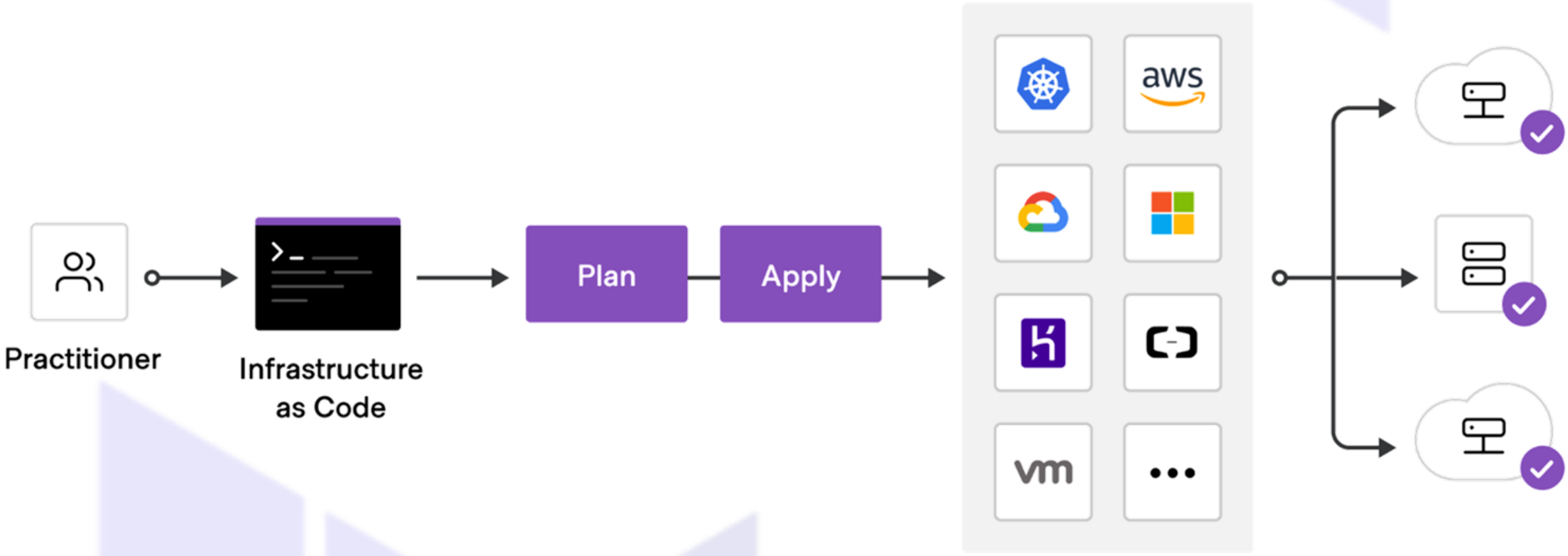
Architecture



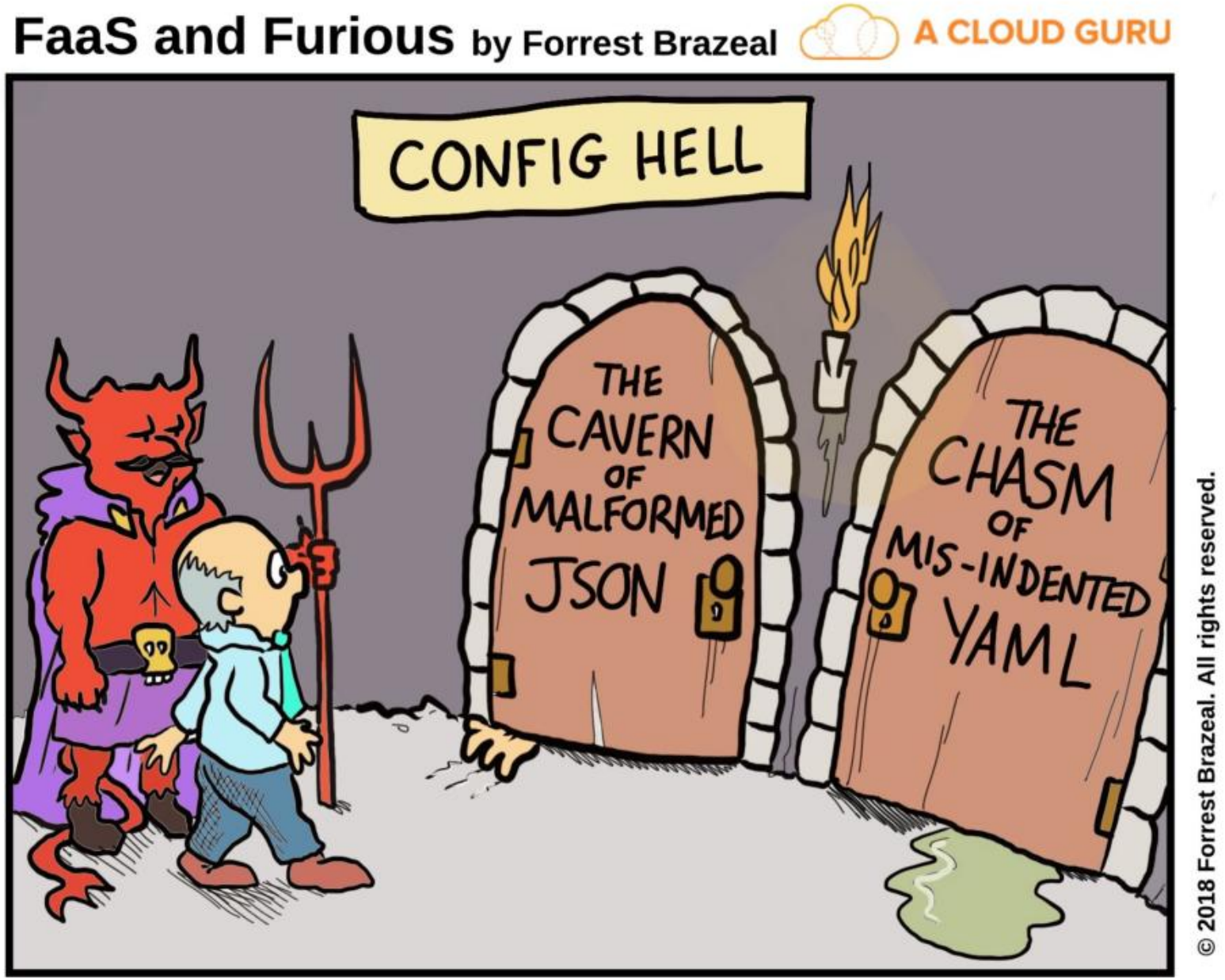
Provides a common language (HCL v2) to orchestrate cloud resources and/or other providers.

Use case

HashiCorp
Terraform

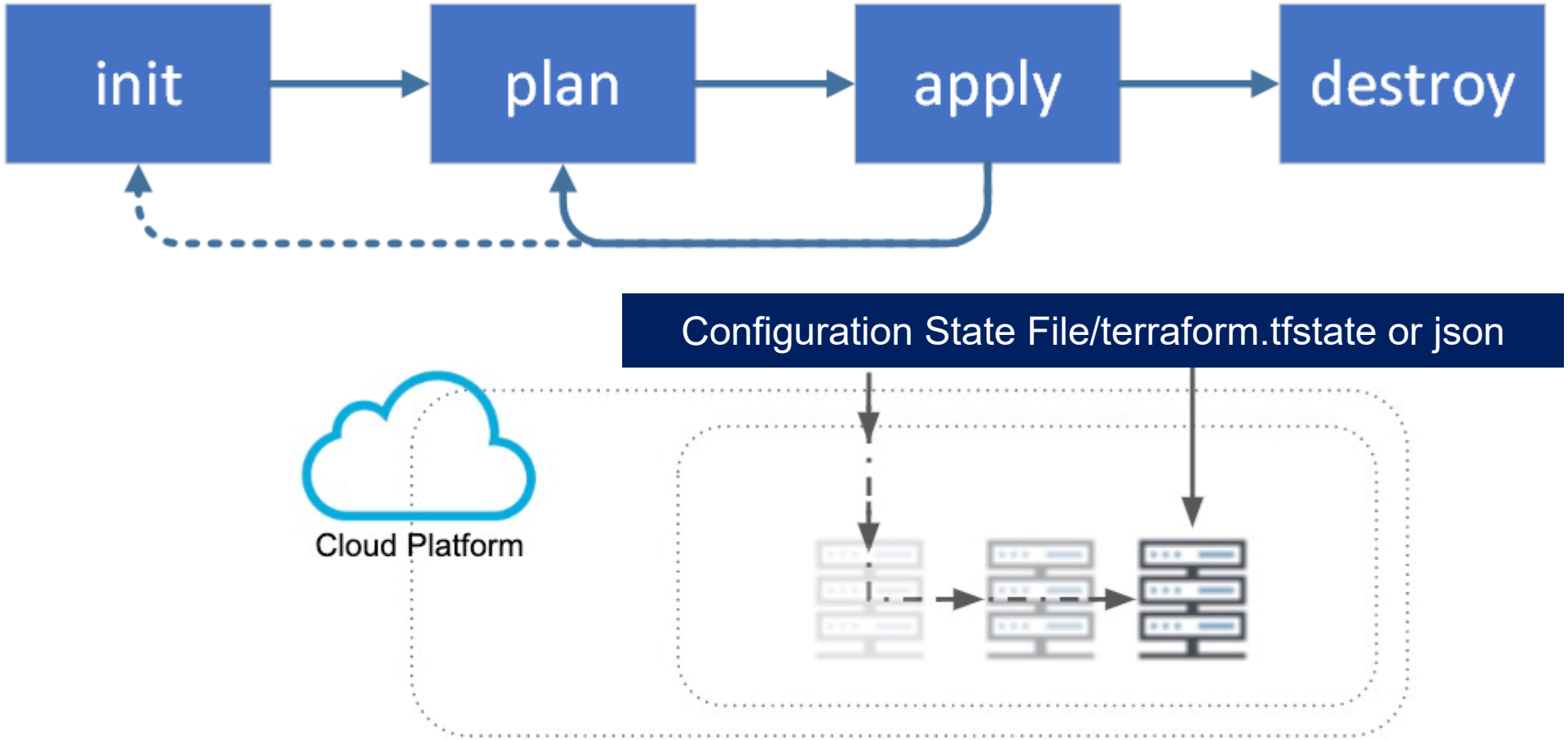


A bit of ..



"Come on, make up your mind -
or it's back to the Sinkhole of Nested XML."

Workflow as example



Terraform is ....

is a command line tool. It is the same for all platforms.
Terraform commands are either typed in manually or run automatically from a script.

Terraform has subcommands that perform different actions.

Basic Terraform Commands

```
terraform version

terraform help

terraform init

terraform plan

terraform apply

terraform destroy
```

Type `terraform subcommand help` to view help on a particular subcommand.

Terraform Code

```
resource aws_vpc "main" {
  cidr_block       = "10.0.0.0/16"
  instance_tenancy = "dedicated"
}
```

Terraform code is based on the HCL2 toolkit. HCL stands for HashiCorp Configuration Language.

Terraform code, or simply `terraform` is a declarative language that is specifically designed for provisioning infrastructure on any cloud or platform.

Terraform init

```
$ terraform init

Initializing provider plugins...
- Checking for available provider plugins...
- Downloading plugin for provider "aws" (hashicorp/aws) 2.35.0...
...
 provider.aws: version = "~> 2.35"

Terraform has been successfully initialized!
```

Terraform fetches any required providers and modules and stores them in the `.terraform` directory. If you add, change or update your modules or providers you will need to run `terraform init` again.

Terraform Plan

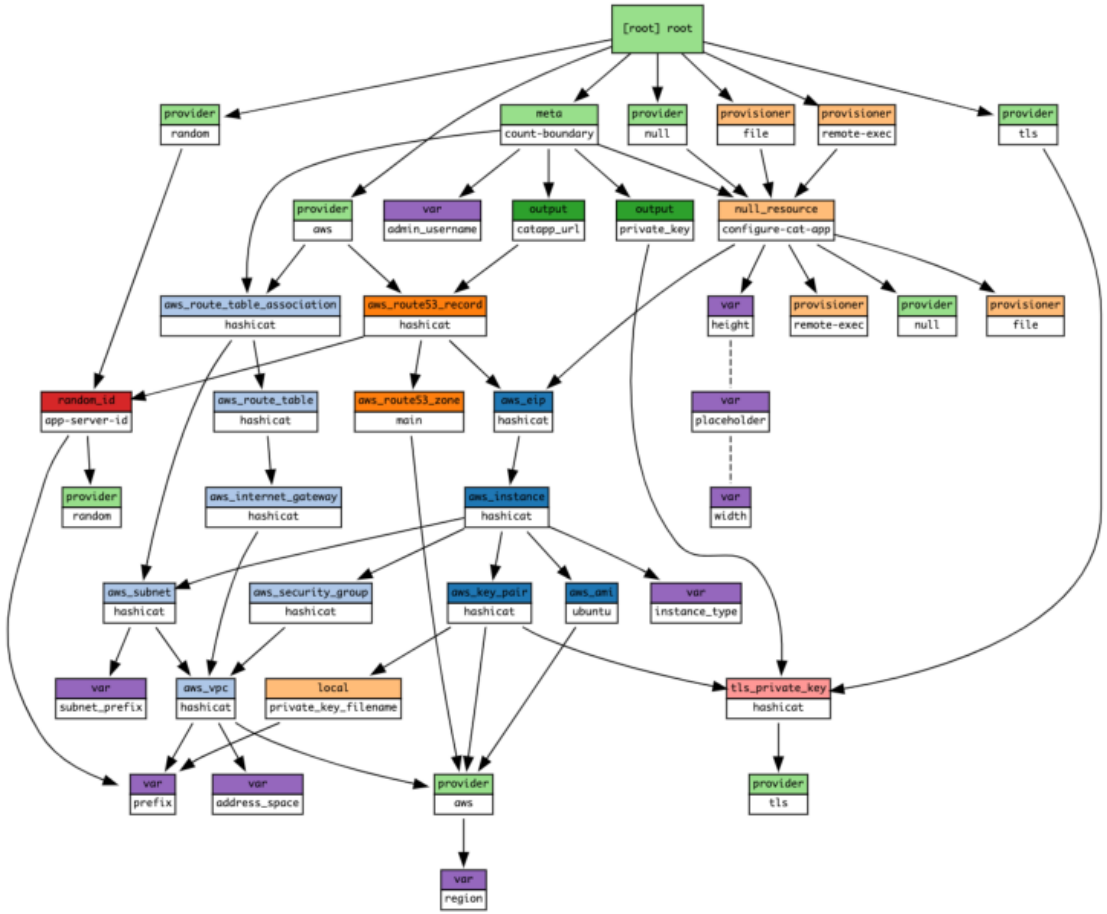
```
$ terraform plan
An execution plan has been generated and is shown below.
Terraform will perform the following actions:
  # aws_vpc.main will be created
  + resource "aws_vpc" "main" {
      + arn                              = (known after apply)
      + cidr_block                       = "10.0.0.0/16"
      ...
      + instance_tenancy                 = "dedicated"
    }
```

Preview your changes with `terraform plan` before you apply them.

Terraform Dependency Graph

The terraform resource graph visually depicts dependencies between resources.

The region and prefix variables are required to create the resource group, which is in turn required to build the virtual network.



Files

terraform.state

STATE FILE

VARIABLES

terraform.tfvars

terraform-provider.tf

CODE

terraform- instances.tf

Depends on your extension, you reorganize the content on files, however it is a good advice to manage a similar structure

Blocks

Providers

Variables

Resource

Data

Outputs

Parameters

Out of scope for this lesson:

Modules

Provisioner

Workspace

Remote backend

Lock

Loops or Conditions

Built-in Variables

Terraform Provider Configuration

The terraform core program requires at least one provider to build anything. You can manually configure which version(s) of a provider you would like to use. If you leave this option out, Terraform will default to the latest available version of the provider.

```
provider "aws" {
  version = "=2.35.0"
}
```

```
- = (or no operator): exact version equality
- !=: version not equal
- \>, >=, <, <=: version comparison
- ~>: pessimistic constraint, constraining both the oldest and newest
version allowed. ~> 0.9 is equivalent to >= 0.9, < 1.0, and ~> 0.8.4
is equivalent to >= 0.8.4, < 0.9
```

Variables

INPUT

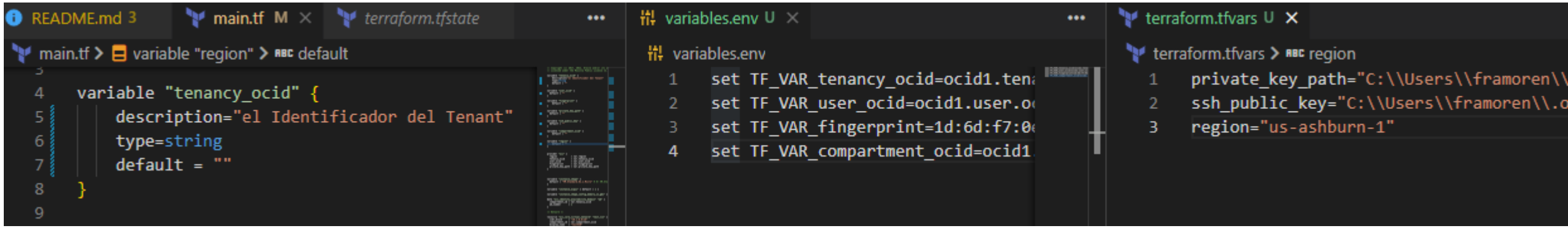
Define variables: See priority next slide.

Types of variables:

* Simple: string, number, bool
* Constructed: list, set, map, object, tuple.
* Validation: regexp, null,

OUTPUT

Attributes: Sensitive



Variables

Terraform variables are placed in a file called variables.tf. Variables can have default settings. If you omit the default, the user will be prompted to enter a value. Here we are declaring the variables that we intend to use.

Priority (high to low)

1. Command line flag - run as a command line switch
2. Configuration file - set in your terraform. tfvars file
3. Environment variable - part of your shell environment
4. Default Config - default value in variables.tf
5. User manual entry - if not specified, prompt the user for entry

Block Structure

```
provider "oci" {
  region           = var.region
  tenancy_ocid     = var.tenancy_ocid
  user_ocid        = var.user_ocid
  fingerprint      = var.fingerprint
  private_key_path = var.private_key_path
}
# See https://docs.oracle.com/iaas/images/
data "oci_core_images" "test_images" {
  compartment_id           = var.compartment_ocid
  operating_system         = "Oracle Linux"
  operating_system_version = "8"
  shape                    = var.instance_shape
  sort_by                  = "TIMECREATED"
  sort_order               = "DESC"
}
/* Network */
resource "oci_core_virtual_network" "test_vcn" {
  cidr_block     = "10.1.0.0/16"
  compartment_id = var.compartment_ocid
  display_name   = "testVCN"
  dns_label      = "testvcn"
}
output "Public_IP_LoadBalanceador" {
  value = "http://${oci_load_balancer_load_balancer.free_lo
}
```

variable
provider
data
resource
output

Module

Folder with code and has similar input and output

Terraform accesses TF files in the current folder only, does not enter to nested folders

How to Provision an AWS Instance

Before we start, we'll need to gather some basic information including (but not limited to):

Instance Name, Operating System (Image), VM Size, Geographical Location (Region), Security Groups

```
resource aws_instance "web" {
  ami           = data.aws_ami.ubuntu.id
  instance_type = "t2.micro"

  tags = {
    Name = "HelloWorld"
  }
}
```

Anatomy of a Resource

Every terraform resource is structured the same way.

resource = Top level keyword

type = Type of resource. Example: `aws_instance .`

name = Arbitrary name to refer to this resource. Used internally by terraform. This field cannot be a variable.

```
resource type "name" {
  parameter = "foo"
  parameter2 = "bar"
  list = ["one", "two", "three"]
}
```

Data Sources

```
data "aws_ami" "ubuntu" {
  most_recent = true
  filter {
    name   = "name"
    values = ["ubuntu/images/hvm-ssd/ubuntu-trusty-14.04-amd64-server-*"]
  }
  filter {
    name   = "virtualization-type"
    values = ["hvm"]
  }
  owners = ["099720109477"] # Canonical
}
```

Data sources are a way of querying a provider to return an existing resource, so that we can access its parameters for our own use.

Terraform Dependency Mapping

Terraform can automatically keep track of dependencies for you. Look at the two resources below. Note the highlighted lines in the `aws_instance` resource. This is how we tell one resource to refer to another in terraform.

You can use depends_on

```
resource aws_key_pair "my-keypair" {
  key_name   = "my-keypair"
  public_key = file(var.public_key)
}

resource "aws_instance" "web" {
  ami           = data.aws_ami.ubuntu.id
  instance_type = "t2.micro"
  key_name      = aws_key_pair.my-keypair.name
```

Terraform State

Terraform is a **stateful** application. This means that it keeps track of everything you build inside of a state file. You may have noticed the `terraform.tfstate` and `terraform.tfstate.backup` files that appeared inside your working directory. The state file is Terraform's source of record for everything it knows about.


```
{
  "terraform_version": "0.12.7",
  "serial": 14,
  "lineage": "452b4191-89f6-db17-a3b1-4470dcb00607",
  "outputs": {
    "catapp_url": {
      "value": "http://go-hashicat-5c0265179ccda553.workshop.aws.hashidemos.io",
      "type": "string"
    },
  }
}
```

Changing Existing Infrastructure

Whenever you run a plan or apply, Terraform **reconciles** three different data sources:

1. What you wrote in your code
2. The state file
3. What actually exists

Terraform does its best to add, delete, change, or replace existing resources based on what is in your *.tf files. Here are the four different things that can happen to each resource during a plan/apply:

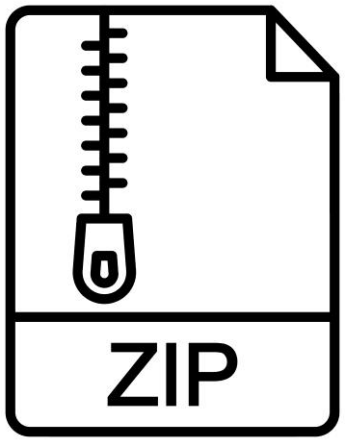
```
+   create
-   destroy
-/+ replace
~   update in-place
```

Best Practices

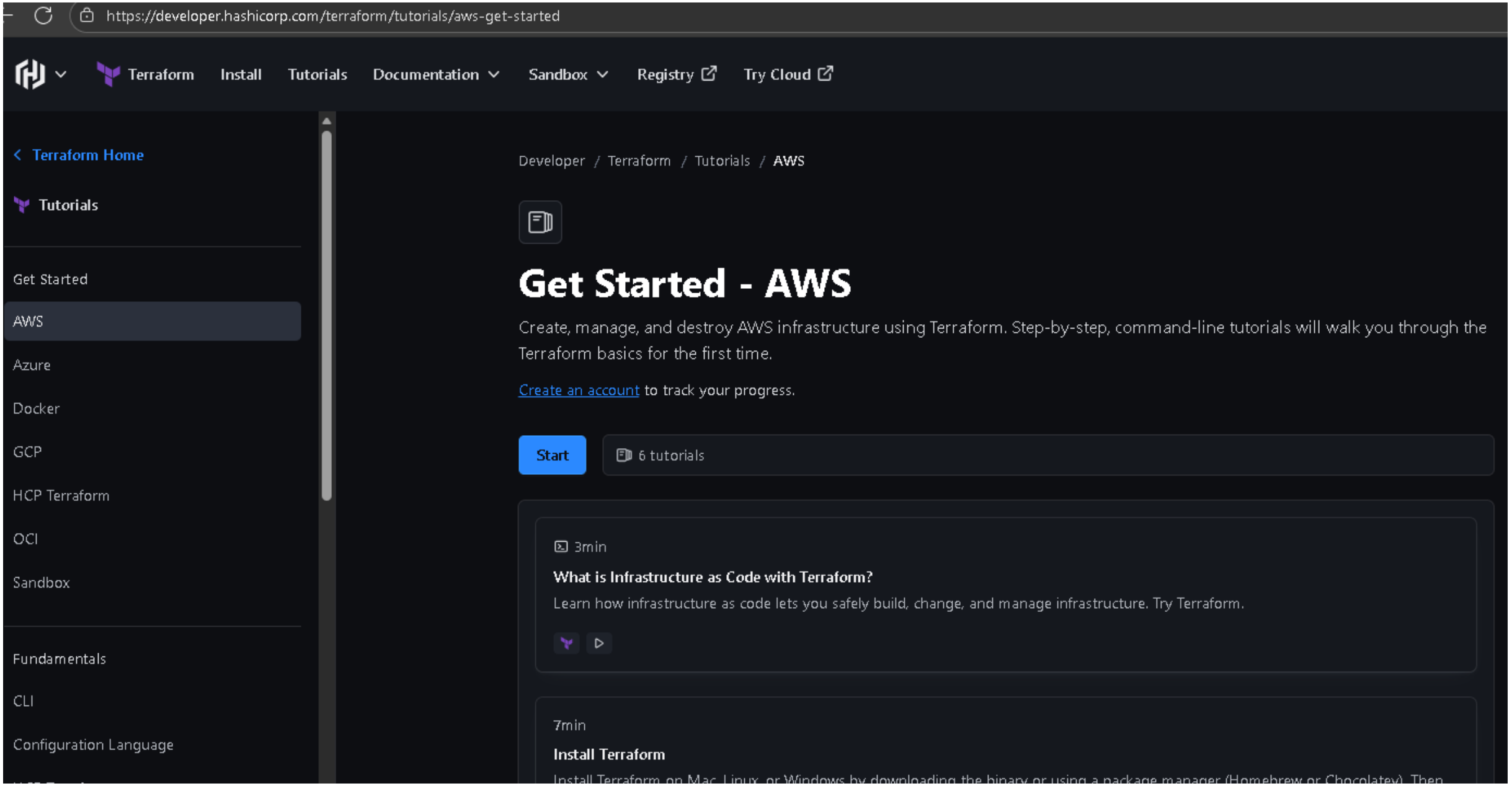
- Keep configurations modular
- Use variables and workspaces for environments
- Never edit the state file manually
- Use `terraform fmt` & `terraform validate`
- Protect credentials (IAM roles, not keys)

Setup

1. Install of terraform and validate
2. Install of AWS CLI and validation –optional-
3. Obtain AWS credentials for CLI
4. Folder with Terraform Code
5. Lifecycle: Init, Plan and Apply
6. Check working environment
7. Changes and Lifecycle
8. End of infrastructure: Destroy



Official Tutorial



References & Resources

- https://developer.hashicorp.com/terraform/docs
- https://registry.terraform.io/providers/hashicorp/aws/latest/docs
- https://learn.hashicorp.com/collections/terraform/aws-get-started
- https://aws.amazon.com/free/