What IAC is?

Infrastructure can be thought of as the resources an application runs on. Traditionally this was composed of servers, storage, and networking. With the advent of virtualization, servers were now split into both physical resources and virtual machines. Cloud providers created additional abstractions, starting with Infrastructure-as-a-Service (IaaS) and moving up to Platform-as-a-Service (PaaS) and Software-as-a-Service (SaaS).

Infrastructure-as-Code (IaC) is the practice of defining and provisioning infrastructure resources through a machine-readable format. The infrastructure provisioned and managed using IaC can include bare-metal servers, virtual machines, networking resources, storage volumes, or even databases provided as a service.

The idea behind the "Infrastructure as Code" IaC is that you write and execute code to define, deploy, and update your infrastructure. This represent a important shift in mindset where we treat all aspects of operations as software, even those aspects that represent hardware (like setting up a server).

Key insight of "DevOps" is that we manage almost everything in code including servers, databases, networks, log files, application configuration, automated tests and so on.

There are four categories of IaC tools:

Ad hoc scripts
Configuration Management Tools
Server Templating Tools
Server Provisioning Tools

1. Ad hoc Scripts

The most easiest approach to automate anything is write a ad-hoc script. Bash Scripts. You take whatever task you were doing manually, break it into steps and define each of those steps in script on your server.

-- set-webserver.sh

```
sudo apt-get update
sudo apt-get install php -y
sudo apt-get install apache2 -y
sudo git clone https://github.com/yourlocation/somerepo.git /var/www/html/
sudo systemctl start apache2
```

Now this is a very simple bash shell script which will start a basic webserver. Good thing about ad-hoc scripts is they are easy.
You can use any popular general purpose programming language and write you code.

Disadvantage of using it -

Whereas IaC tools provides concise APIs for doing complicated tasks. If you are using general purpose programming language, you have to write completely custom code for everything. And further, IAC tools usually enforce a particular structure for your code, whereas using bash shell script, each developer can use his/her own style and do something different.
This doesn't look a big problem for the previous five line script. But it will get messy if you try to use ad-hoc scripts to manage galore of servers, databases etc
Ad-hoc scripts are good for small one-off tasks, but if your intent to manage your infrastrucure as code, then they should be avoided

2. Configuration Management Tools

Ansible, Puppet, Chef & SaltStack they all are configuration management tools. Means they are designed to install and manage software on existing servers. Like here is a small ansible playbook which will install the same webserver

```
## set-webserver.yaml
- name: install php & apache
apt:
name:
- php
- apache2
- name: copy code from repo
git: repo=https://github.com/yourlocation/somerepo.git dest=/var/www/html/
- name: start apache
systemd:
name: apache2
state: started
enabled: yes
```

Code looks similar to bash script, but using Ansible offers a lot of advantages:

Coding Conventions: Ansible enforces a consitent, predictable strucure, clearly named parameters and so on. So every develepor has to use the same conventions that makes it easier to read the code.
Idempotence: Writing a ad-hoc script that works correctly even if you run it again and again is difficult. Say every time you go to create a user, you need to remember to write logic in your script to first check if the user already exists. Adding more lines of code. The code that run correctly no matter how many times you run it is called "idempotent code".
Distribution: Ad-hoc scripts are designed to run on a single machine. Ansible and other configuration managemtn tools are designed to manage large number of remote servers

3. Server Templating Tools

An alternative to configuration management is "server templating tools" like Docker, Packer, Vagrant etc. Instead of launching some servers and then configuring them by running the same code on every machine. The idea is to create a image (golden image) of a server that has all the things we need in the OS. You can then use this image on all your servers using other IaC tools.

There are different server templating tools for different purposes.
Packer is normally used to create image that you run directly on top of production servers. Like the AMI you use in your AWS account.
Vagrant is used to create images that you run on your development computers. Like the virtual box image your run.
Docker is used to create images of individual applications. You can run Docker images on production or development computers, so long as some other tool as configured that computer with the Docker Engine.

Server templating tools shift the infra to "**immutable infrastructure**".
The idea is once you've deployed a server, you never make changes to it again. If you need to update something, like new version of application or new version of your code, you create a new image from your server template and you deploy it on a new server

4. Server Provisioning Tools

Like configuration management tools like Ansible, Puppet & Chef were responsible for the code that runs on each server. Server provisioning tools like Terraform, CloudFormation, OpenStack Heat are responsible for creating the servers themselves. Not only servers, you can create databases, load balancers, subnets, firewall, routing rules and almost every component of your infrastructure. This terraform code deploy a webserver

Benefits of IAC
When your infrastructure is defined as code, you can use wide variery of software practices and this will improve your software delivery process
Self Service - We normally have a smaller number of sysadmins. This becomes bottleneck when company grows. If your infrastructure is defined as code, then the entire process can be automated.
Documentation - Instead of the state of your infrastructure locked in sysadmin's head, you can represent the state of your infrastructure in a file that anyone can read. So things won't stop if sysadmin is on leave or he has left.
Version Control - You can store your IaC files in version control
Validation - If the infrastructure is in code, then for every single change, you can perform a code review, run a suite of automated tests and pass the code through static analysis tools.
Reuse - You can package your infrastructure as reusable modules

How Terraform Work

Terraform is a open source tool created in Go programming language. The code compiles into a single binary, called terraform. Terraform binary makes API calls on your behalf to one or more providers, like AWS, Azure, Google etc.

How terraform knows what API call to make and to which provider. This is taken from your terraform configurations.

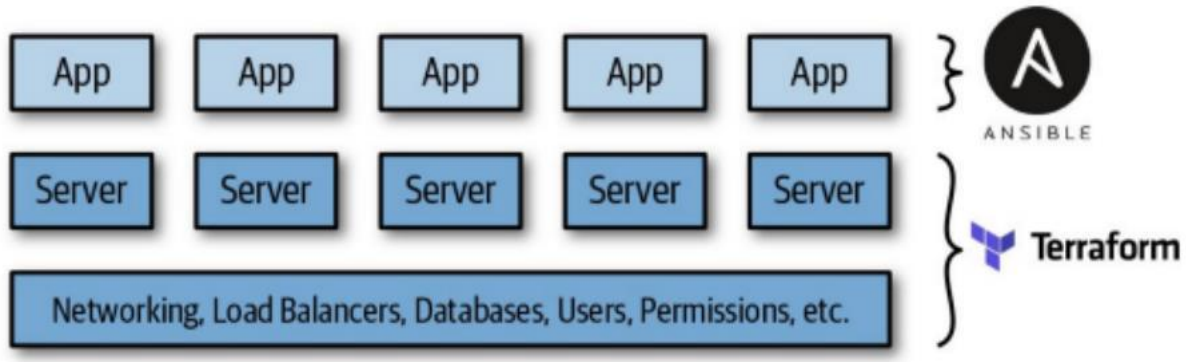
```
## sample.tf
resource "aws_instance" "example" {
ami ="ami-12345678"
instance_type ="t2.micro"
}
```

MULTIPLE TOOLS TOGETHER

common combinations that many companies uses :

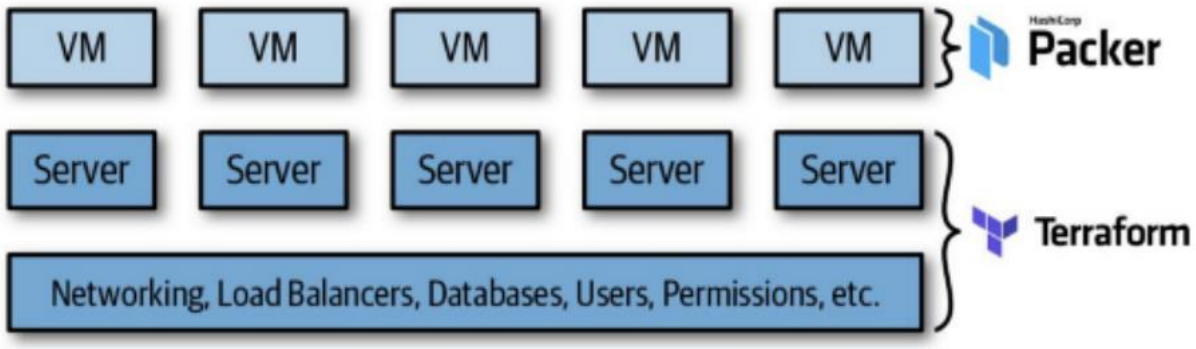
Provisioning plus configuration management

Terraform and Ansible - Use Terraform to deploy all the underlying infrastructure, including the network topology (i.e., virtual private clouds [VPCs], subnets, route tables), data stores (e.g., MySQL, Redis), load balancers, and servers. Then use Ansible to deploy your apps on top of those servers



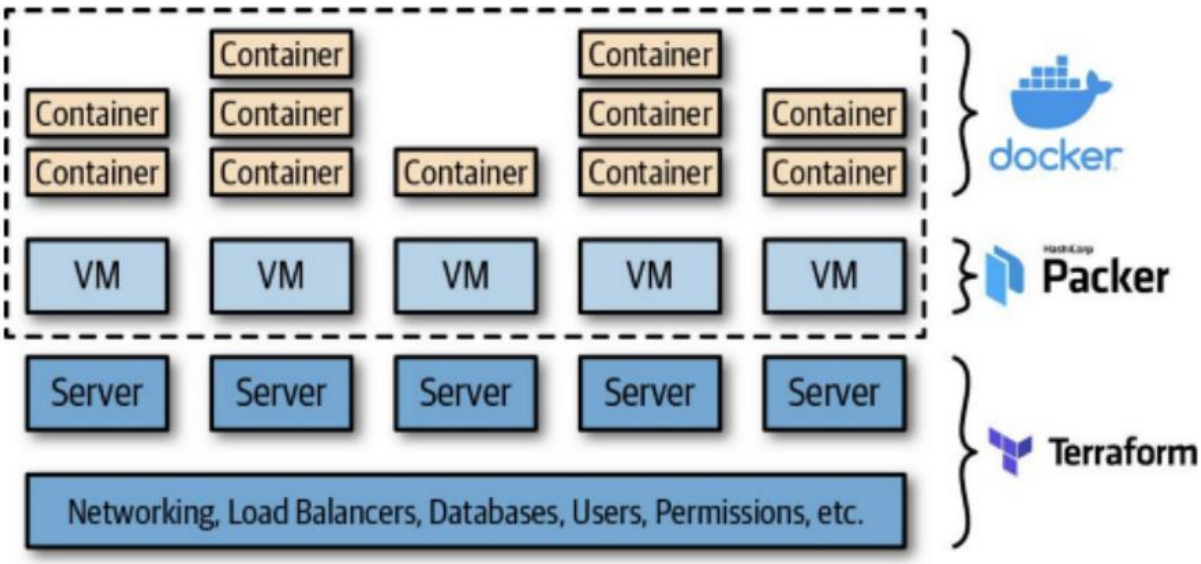
Provisioning plus server templating

Terraform and Packer - Use Packer to package your apps as VM images. Use Terraform to deploy (a) servers with these VM images and (b) the rest of your infrastructure, including the network topology (i.e., VPCs, subnets, route tables), data stores (e.g., MySQL, Redis), and load balancers

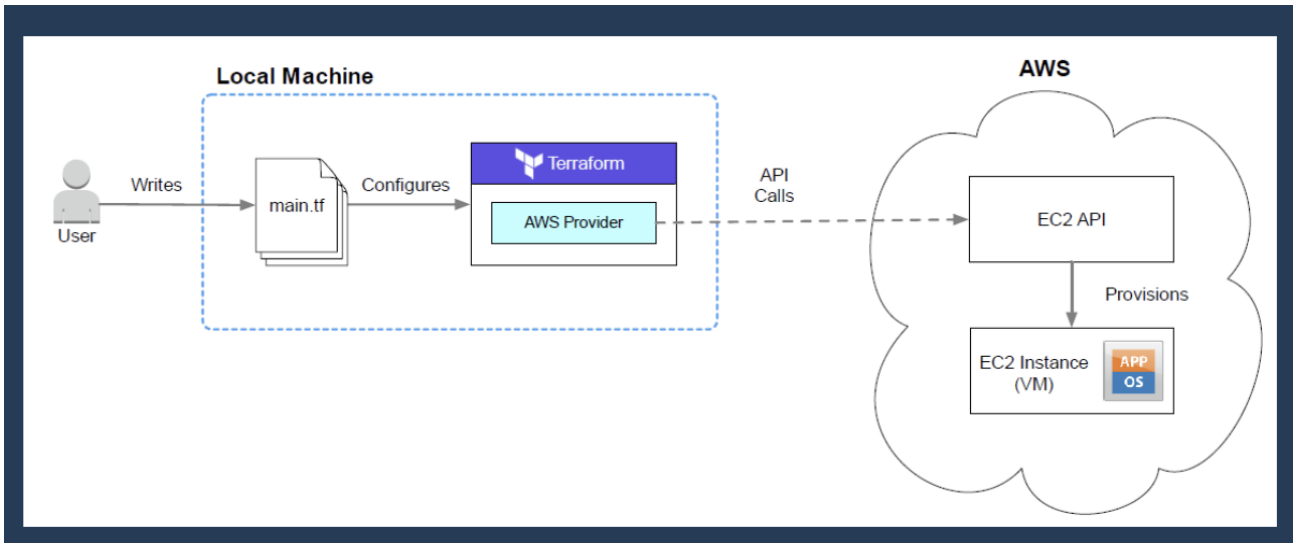


Provisioning plus server templating plus orchestration

Terraform, Packer, Docker, and Kubernetes - Use Packer to create a VM image that has Docker and Kubernetes installed. Then use Terraform to deploy
(a) a cluster of servers, each of which runs this VM image, and
(b) the rest of your infrastructure, including the network topology (i.e., VPCs, subnets, route tables), data stores (e.g., MySQL, Redis), and load balancers.
Finally, when the cluster of servers boots up, it forms a Kubernetes cluster that you use to run and manage your Dockerized applications



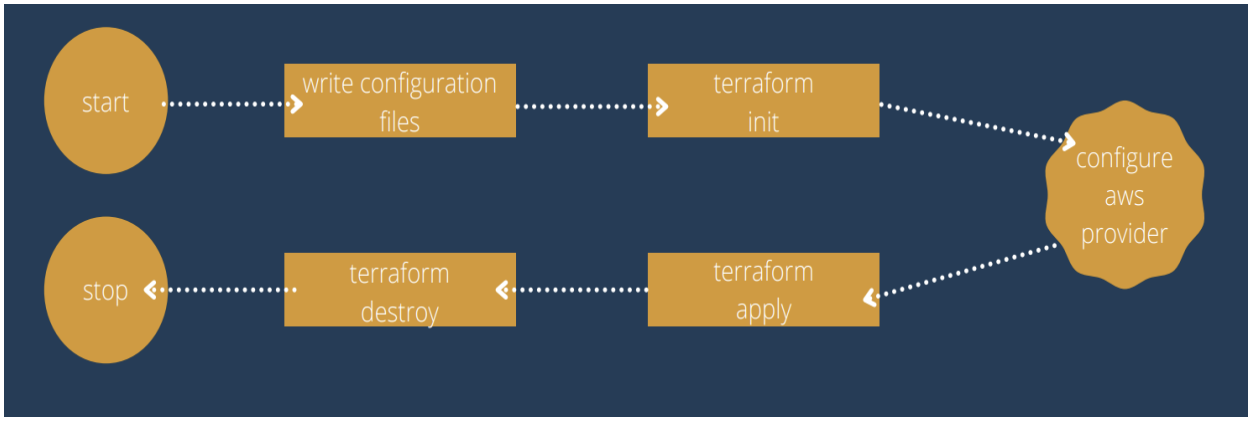
Starting with Terraform



Deploying a virtual machine (EC2 instance) onto AWS. We'll use the AWS provider for Terraform to make API calls on our behalf and deploy an EC2 instance. Finally, we'll have Terraform terminate the instance.

The steps we'll take for deploying the project are:

1. Writing Terraform configuration files
2. Configuring the AWS provider - install AWS CLI
3. Initializing Terraform with - terraform init
4. Deploying the EC2 instance with - terraform apply
5. Cleaning-up with - terraform destroy



Authenticating to AWS

Usually we don't pass secrets into the provider as plain text, especially when this code will later be checked into version control, so many providers have the ability to read environment variables or shared credential files to retrieve this secret information dynamically. We will check this later.

provider "aws" {

Region ="us-east-2"
Access_key= "your access key"
Secret_key= "your secret key "

}

Terraform Providers & Resources

Terraform supports multiple providers.
We have to specify the provider details for which we want to launch the infrastructure for.
With Providers, we also have to add the tokens which will be used for authentication

provider "aws" {

Region ="us-east-2"

}

Resources are the refence to the individual services which the provider has to offer
eg : resource "aws_instance", resource "aws_alb", resource "aws_security_group"
resource "aws_instance" "example" {

```
        ami = “”
        instance_type = “”
}
```

Initializing Terraform

We first have to initialize the workspace. Even though we have declared the AWS provider, Terraform still needs to download and install the binary from the provider registry. Initialization is required for all new workspaces, as well as any new Terraform project you download from source control and are running for the first time. *terraform init* is the first command you can and must run on a Terraform configuration You can initialize Terraform by running the command:

terraform init

Now we’re ready to deploy the EC2 instance using Terraform. Run the command:

terraform apply

You can verify your resource was created by finding it in the AWS console for EC2

Terraform Destroy

Cleaning up of the infrastructure.
You always want to take down any infrastructure you are no longer using, as it costs money to run stuff in the cloud. Terraform has a special command to destroy all resources.

#terraform destroy

If you need to destroy only a specific resource then use “-target”

#terraform destroy -target aws_instance.example

Terraform State File

All of the stateful information about the resource is stored in a file called terraform.tfstate. This state allows terraform to map real world resources to your existing configuration.
The terraform show command can be used to print human readable output from the state file and makes it easy to list all the data of the resources Terraform manages.

terraform show

WARNING!!

Never manually edit or delete the terraform.tfstate file, or else Terraform will lose track of all managed resources

Desired State and Current state

Desired state : EC2 = instance_type = “t2.micro”
State which is mentioned inside the resource block.

Current state : the details stored in the tfstate file represents the current state. It looks into what exactly is running on the infrastructure
eg: “instance_state” : “stopped”, && “instance_type”: ”t2.nano”

terraform plan will always make sure that the Desired state == current state

Multiple Instance of a provider

There may be times when you need to specify multiple instances of the same provider. For instance, the AWS provider is region specific. If you want to create AWS resources in more than one region, you’re going to need multiple instances of the provider.

All providers support the use of the **alias** meta-argument, which enables you to give a provider an alias for reference. When creating resources using that provider, you can specify the alias of the provider instance to use

```
provider "aws" {
    region = "us-east-1"  #virginia region
    alias = "usa"
}
```

Provisioners

When a resource is created, you may have some scripts or operations you would like to be performed locally or on the remote resource. Terraform provisioners are used to accomplish this goal.

What are the use cases for a provisioner anyway?

Loading data into a virtual machine
Bootstrapping a VM for a config manager
Saving data locally on your system

Types of Provisioners:

Local_exec

local_exec is used to run a script/command on local machine where we are running our terraform code

```
resource “aws_instance” ”web”{

 ###
}

provisioner “local_exec”{
        command=”echo  Server IP address is ${aws_instance.web.private_ip}”
}}
```

Remote_exec

The remote-exec provisioner invokes a script on a remote resource after it is created. This can be used to run a configuration management tool, bootstrap into a cluster, etc.

Inline-> It is used to execute list of commands.

```
provisioner "remote-exec" {
    inline = [
     "sudo amazon-linux-extras install -y nginx1.12",
      "sudo systemctl start nginx",
   ]
}
```

Outputs

Terraform can make outputs available after a configuration has been successfully deployed. The outputs created by a root module (that’s your main configuration) are printed as part of the console output and they can be queried using the terraform output command.

Creating an output in a configuration follows a simple format. The output needs a name and value

```
Output “myoutput”{
        value=aws_instance.ami
        description=”The ami value of the instance”
}
```

The output block can have two additional arguments: **sensitive** and **depends**_on

If an output has sensitive set to true, Terraform will not print the value of the output at the CLI or include it in logging. If the source of the value is already defined as sensitive, then the sensitive argument must be set to true.

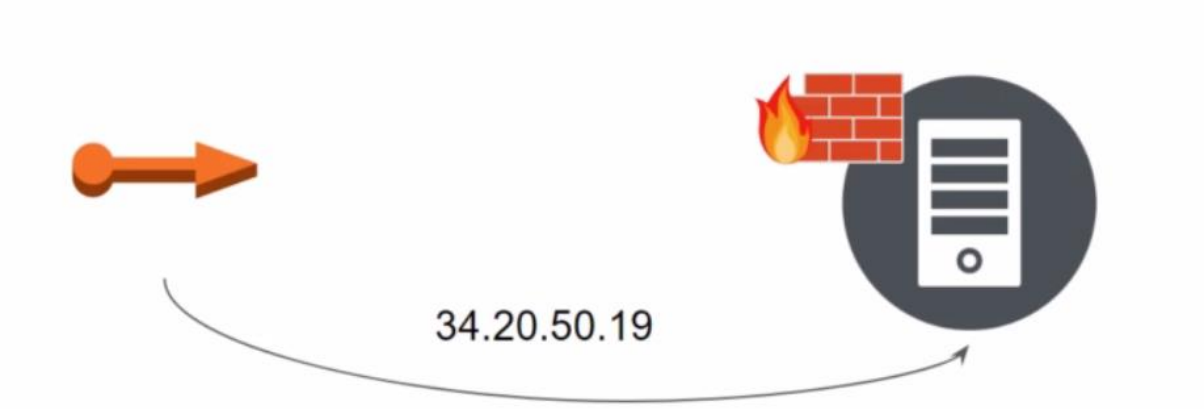
```
output "instance" {
        value = aws_instance.web
        description = "Web instance"
        # Won't show in the cli
        # Still visible in state data
        # Could be used in a module
        sensitive = true
}
```

The depends_on argument takes a list of resources the output is dependent on before being rendered. Usually Terraform can determine dependence on its own, but there are exceptional cases where the chain of dependence is not obvious.

Outputs are rendered when terraform apply is run. If you change your existing outputs, or add new ones, you'll need to run apply again even if nothing else has changed in the configuration

Attributes and References

An outputted attributes can not only be used for the user reference but it can also act as a input to other resources being created via terraform.



Input Variables

Variables are defined in a Terraform configuration. You can supply values for those variables at runtime, or set a default value for the variable to use. If you do not set a default value for a variable, Terraform will require that you supply one at runtime. Failing to supply a value will cause the relevant command to error out.

```
variable "aws_region" {
        type = string
        default ="ap-south-1"
        description ="The AWS region to use for this configuration"
}
```

You can also set a variable as sensitive, which means that the value stored in the variable will not be shown in cleartext in logging, terminal output, or plan files.

Setting environment variables with the prefix TF_VAR_ followed by the variable name.
TF_VAR_instancetype m5.large

Placing a terraform.tfvars or terraform.tfvars.json file in the same directory as the configuration.
Placing a file ending with .auto.tfvars or .auto.tfvars.json in the same directory as the configuration

Count and Count Index

The count parameter on resource can simplify configurations and let you scale resources by simply incrementing a number. With count parameter specify the count value and the resource can be scaled up.

```
resource ”aws_instance” “foo-1”{

        ami=”ami-0484464er45215fgop”
        instace_type = “t2.mirco”
        count = 3

}
```

In resource block where count is set, an additional count object is available in expression, so you can modify the configuration of each instance

count.index – the distinct index number (starting with 0)

```
resource ”aws_iam_user” “lb”{
        name=”loadbalancer.${count.index}”
        count=5
        path=”/system/”
}
```

Format (fmt)

The terraform fmt command is used to rewrite Terraform configuration files to match the canonical format and style outlined by the Terraform language style conventions. Recommend running terraform fmt after updating to a newer version of Terraform.

The fmt command has the syntax - #terraform fmt [options] [DIR].

The fmt command scans the current directory for configuration files ending in tf and tfvars unless a DIR argument is specified. There are several options available for the fmt command including: list, write, diff, check, and recursive. All of these are optional, and further information about each can be found by running.

Taint

The terraform taint command does exactly what you might expect, it taints a resource in the state. Terraform will destroy and recreate the resource the next time the configuration is applied. Running terraform taint on its own will not destroy the resource immediately, it simply edits the state. By running terraform plan on the root module, you will see that a resource has been tainted and will be recreated on the next apply. Running terraform apply will execute the plan, destroying then recreating the resource. The taint command has the syntax

#terraform taint aws_instance.mywebserver

Recreating a resource is useful for situations where a change is not obvious to Terraform, or the resource’s internal state is known to be bad. Terraform does not have visibility into the inner state of some resources, for instance a virtual machine. As long as the virtual machine’s resource attributes seem valid, Terraform will not flag it for recreation. As an example, you may make a charge to the startup script for a virtual machine that will not take effect until the machine is recreated. Terraform does not track the startup script as a resource attribute, so it has no knowledge that something about the resource has changed. By tainting the virtual machine resource, Terraform will know that the resource must be replaced. Other resources in the configuration may have properties that are dependent on the resource being recreated, therefore a tainted resource may result in additional changes in the remainder of the configuration. For instance, a load balancer connected to the virtual

machine will need to be updated with the new virtual machine's network interface to direct traffic properly.

Import

The terraform import command is used to import existing resources into Terraform. This enables you to take existing resources and bring them under Terraform's management. Most environments are not going to be greenfield deployments with no existing infrastructure. The import command is a way to bring that existing infrastructure under Terraform's management without altering the underlying resources. The import command can also be useful in situations where an administrator manually created a resource outside of Terraform's management & that resource needs to be brought into the fold. Ideally, once you've adopted Terraform, all new resources will be created and managed through Terraform.

As of this writing, the current version of Terraform does not generate the configuration for an imported resource, it simply adds the resource to the state. It is up to you to write a configuration that matches the existing resources prior to running the import command. Admittedly, this is an inefficient process, but there are benefits

So for now, Terraform import can bring existing resources into management by Terraform. Import alone does not create the necessary configuration to import a resource, that must be done manually.

How does Import Work ?

Create a EC2 instance, manually using dashboard in your AWS account. Note down the instance's ID.

Create a .tf file with these resource

```
resource "aws_instance" "fromdash" {
        # resource arguments
}
```

Run - terraform import aws_instance.fromdash <instance-ID>

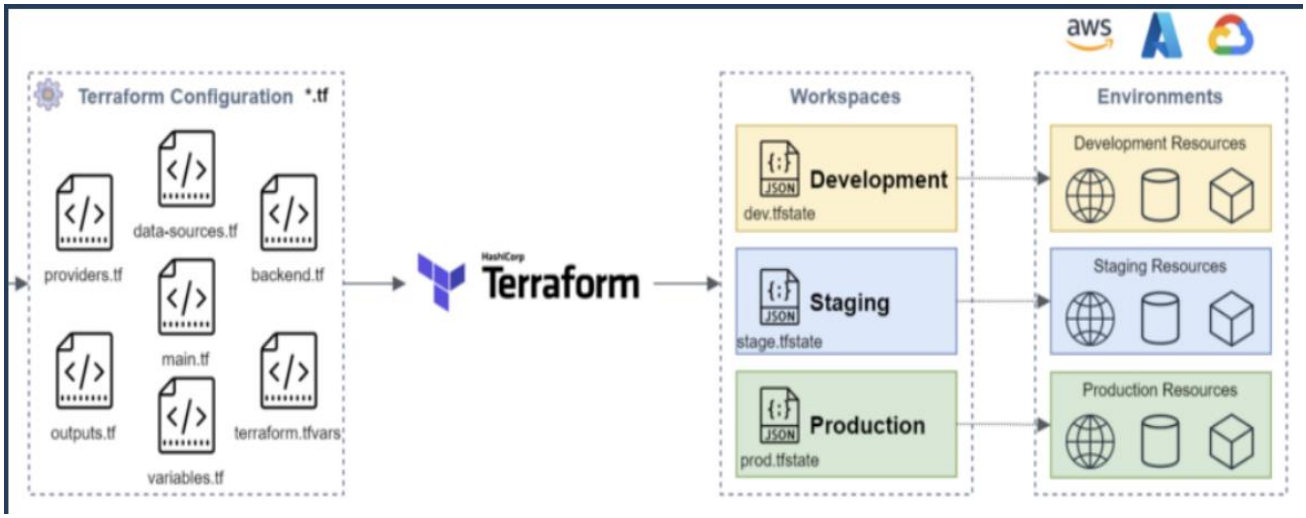
Workspace

One common need on infrastructure management is to build multiple environments, such as testing and production, with mostly the same setup but keeping a few variables different, like networking and sizing.

Terraform workspace allows you to create different and independent states on the same configuration. And as it's compatible with remote backend this workspaces are shared with your team.

Terraform workspaces allow you to store your Terraform state in multiple, separate, named workspaces. Terraform starts with a single workspace called "default," and if you never explicitly specify a workspace, the default workspace is the one you'll use the entire time.

Workspaces allows you to separate your state and infrastructure without changing anything in your code when you wanted the same exact code base to deploy to multiple environments without overlap. i.e. Workspaces help to create multiple state files for set of same terraform configuration files



#terraform -help workspace

Workspaces are managed with the terraform workspace set of commands. Below is a list of subcommands available for use

• new - Create a new workspace and select it

• select - Select an existing workspace

• list - List the existing workspaces

• show - Show the name of the current workspace

• delete - Delete an empty workspace

To run terraform workspace delete, the workspace must not currently be selected and the state file must be empty. The -force flag can be specified if the state file is not empty, and the resources created by the state file will be abandoned.

Workspaces can be configured to use remote backend like AWS - S3

State

The state maintained by Terraform is a JSON document stored either locally or on a remote backend. Terraform users are highly discouraged from making direct edits to the file.

There are some occasions where you will need to directly interact with the Terraform state, and in those cases you will use the terraform state command and its subcommands to assist you. terraform state has a set of subcommand commands.

list - List resources in the state
mv - Move an item in the state
pull - Pull current state and output to stdout
push - Update remote state from a local state file
rm - Remove instances from the state
show - Show a resource in the state

List

The terraform state list command is used to list resources within a Terraform state. The list subcommand has the following syntax:

#terraform state list [options] [address...]

The options allow you to specify a particular state file using -state=statefile or a specific id using -id=ID. The address refers to a particular resource within the state file as described in the preceding Resource Addressing section. If no address is given, then all resources will be returned by the command

Move

The terraform state mv command is used to move items in a Terraform state. The mv subcommand has the following syntax:

#terraform state mv [options] SOURCE DESTINATION

There are a few reasons to use the mv command. If an existing resource needs to be renamed, but you don't want to destroy and recreate it, you can update the name in the configuration and use the mv command to "move" the resource to its new name.

Let's say that we want to change out the name label for the resource instance

aws_instance.nginx to aws_instance.web. First we would update the relevant code from this resource

"aws_instance" "nginx" to this resource "aws_instance" "web"

Run terraform plan without updating the state. After that run the mv command.

terraform state mv aws_instance.nginx aws_instance.web
Run terraform plan again, to confirm that infrastructure is updated

Pull

The terraform state pull command is used to manually download and output the state from a remote backend. This command also works with the local backend.

The pull subcommand has the following syntax:

#terraform state pull [options]

This command is equivalent to viewing the contents of the Terraform state directly, in a read-only mode. The output of the command is sent in JSON format to stdout and can be piped to another command - like jq - to extract data.
The primary use for the command is to read the contents of remote state, since if you were using the local backend you could just as easily use cat on the local file

Push

The terraform state push command is used to manually upload a local state file to a remote backend. This command also works with the local backend. You probably won't need to use this command. If you think you do, take a moment and reconsider. You are taking a local file and replacing the remote state with the contents of that local file. VERY DANGEROUS !

The push subcommand has the following syntax:

#terraform state push [options] PATH

Remove

The terraform state rm command is used to remove items from the Terraform state. The rm subcommand has the following syntax:

#terraform state rm [options] ADDRESS

The most common use for the rm command is to remove a resource from Terraform management. Perhaps the resource will be added to another configuration and imported. Perhaps Terraform is being replaced by something else. In either case, the goal is to remove the reference to the resource without deleting the actual resource.

Let’s say that we are no longer going to manage our aws_instance with Terraform.
First we can run the rm command.

terraform state rm aws_instance.web
Removing the resource from the state file also does not remove it from the configuration. The resource and all references should be removed from the configuration before running another plan or apply. Otherwise Terraform may

attempt to create the resource again or throw errors due to invalid references in the configuration file.

Show

The terraform state show command is used to show the attributes of a single resource in the Terraform state. The show subcommand has the following syntax:

#terraform state show [options] ADDRESS

The ADDRESS specified by the show command must be an instance of a resource and not a grouping of resources created by count or for_each. The attributes of the resource are listed in alphabetical order and are sent to stdout for consumption by some type of parsing engine.

Modules

Terraform module is a set of Terraform configuration files (*.tf) in a directory. The advantage of using modules is reusability. You can use the terraform module available in the terraform registry or share the module you created with your team members.

A Terraform module allows you to create logical abstraction on the top of some resource set. In other words, a module allows you to group resources together and reuse this group later, possibly many times.

Example:

main.tf: This is our main configuration file where we are going to define our resource definition.
variables.tf: This is the file where we are going to define our variables.
outputs.tf: This file contains output definitions for our resources.

Workflow

Write

Writing Infrastructure as Code (IaC) is just like writing any other piece of code. The code you are writing for Terraform will either be in HashiCorp Configuration Language (HCL) or in Javascript Object Notation (JSON). It is far more common to write in HCL, since it is simpler for a human to write and read.

The code you write will often be checked into some type of Source Control Management (SCM) repository where it can be versioned and changes are tracked. This is not a hard requirement, but is a best practice. Whenever we start a new project in Terraform, we first create a source code repository to commit our code to. This serves the purpose of tracking changes, backing the code up outside of our computers, and versioning so we can roll-back when we mess something up. Using source control becomes even more important when working on a project as a team.

Plan

During the plan process, you check to see what changes your code will make in the target environment. You might know what you want the end result to be, but very few of us get it right the first time. The planning process allows you to test your changes and make sure that the changes you had in your mind match up with the reality of what Terraform will do.

During your writing process, you will often run terraform plan to preview changes and make sure you are on the right track. By doing so iteratively, you can ensure your configuration remains valid through each small change. All of us have tried to write a big change into a configuration before, and then spent hours trying to figure out where we went wrong. You can save yourself a lot of time by keeping your changes small, and making sure that the plan is valid on a rolling basis. Each time you run a plan you have the option to save the proposed plan to a file. When you are sure that your configuration is ready to deploy, you can run terraform apply which will show the proposed changes once more and ask for your approval to implement the change

Apply

The last step in the workflow is to apply your updated configuration to the target environment and make you dreams a reality. With a successful apply of the new configuration, it’s also a good time to commit your changes to source control to make sure they are not lost in the fuss. Then it’s time to start the whole process over again! Infrastructure requirements are constantly changing, and you’ll need to keep altering the configuration, planning its changes, and applying them to the target environment. It’s a cycle of continuous improvement, just like any other software development lifecycle

Terraform Validate

The terraform validate command is used to validate the syntax of Terraform files. Terraform performs a syntax check on all the Terraform files in the directory specified, and will display warnings and errors if any of the files contain invalid syntax.

This command does not check formatting (e.g. tabs vs spaces, newlines, comments etc.).

Validation can be run explicitly by using the terraform validate command or implicitly during the plan or apply commands. By default, terraform plan will validate the configuration before generating an execution plan. If terraform apply is run without a saved execution plan, Terraform will run an implicit validation as well.

The configuration must be initialized before validation can be run. Terraform is trying to validate the syntax of a configuration, including any modules, providers, and provisioners. The plugins for each of those resources are downloaded during initialization, and Terraform needs those plugins to understand and validate the syntax of a configuration.

Terraform Plan

The terraform plan command is used to create an execution plan. Terraform performs a syntax validation of the current configuration, a refresh of state based on the actual environment, and finally a comparison of the state against the contents of the configuration.

Running the plan command does not alter the actual environment. It may alter the state during the refresh process, if it finds that one of the managed resources has changed since the previous refresh. Running plan will show you whether changes are necessary to align the actual environment with the configuration, and what changes will be made.

The execution plan generated by Terraform can be saved to a file by using the -out argument and giving a file name as a destination. The execution plan is aligned with the current version of the state, and if the state changes Terraform will no longer accept the execution plan as valid. The saved execution plan can be used by terraform apply to execute the planned changes against the actual environment

Terraform Apply

The terraform apply command executes changes to the actual environment to align it with the desired state expressed by the configuration or from the execution plan generated by terraform plan.

By default, apply will look for a Terraform configuration in the current working directory and create an execution plan based on the desired state and the actual environment. The

execution plan is presented to the user for approval. Once the user approves the execution plan, the changes are applied to the actual environment and the state data is updated.

If a saved execution plan is submitted to terraform apply, it will skip the execution plan generation and approval steps, and move directly to making the changes to the actual environment and state data.

You do not have to run terraform plan before terraform apply. When running Terraform manually outside of a team setting, skipping the plan stage might be justified. If you start to automate Terraform runs, or collaborate with a team, then it is advised to always run terraform plan first and save the execution plan to a file.

The terraform apply command has several arguments. The most commonly used ones are listed:

-auto-approve: skips the approval step and automatically moves to making changes
-input: determines whether or not to prompt for input, set to false in automation
-refresh: whether or not a refresh of state should be run, defaults to true
-var: set a value for a variable in the configuration, can be used multiple times
-var-file: specify a file that contains key/value pairs for variable values

Terraform Destroy

Destroying what you deployed may not seem like an obvious step in the lifecycle of infrastructure, but there are many cases where this might be useful. You might be deploying development environments for a project and you want to clean up the resources when the project concludes.

You might be building an environment for testing in a CI/CD pipeline, and want to tear down the environment when the tests are complete. The terraform destroy command is used to destroy Terraform-managed infrastructure.

The destroy command is obviously a powerful command. It will delete all resources under management by Terraform, and this action cannot be undone. For that reason, Terraform will present you with an execution plan for all the resources that will be destroyed and prompt to confirm before performing the destroy operation