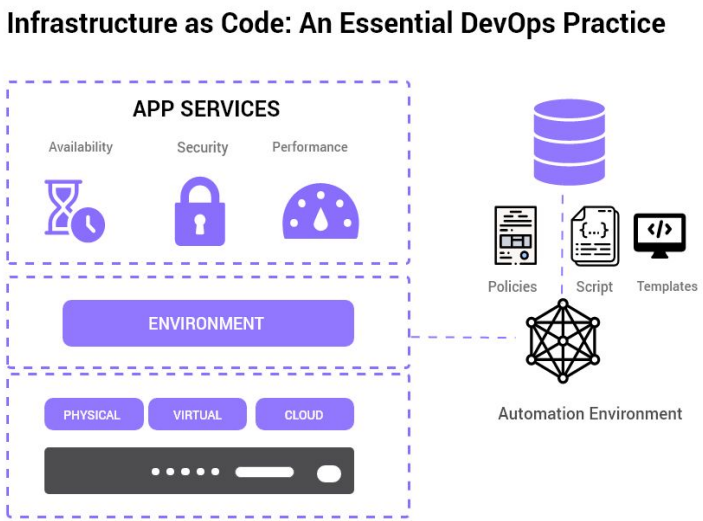
1) **What is Infrastructure as Code (IaC)**
2) **Infrastructure as code (IaC) benefits**
3) **What is Terraform?**
4) **Terraform Features**
5) **Terraform Basic Commands**
6) **Terraform Main Commands In-depth explanation**
   Terraform get, init, validate, plan, apply, state, workspaces, destroy, import, taint, graph
7) **Terraform Coding**
   Structure, resources, data sources, variable, providers, functions, modules, provisioners, Lifecycle Policy, Remote State backend
8) **Terraform Cloud & Enterprise**

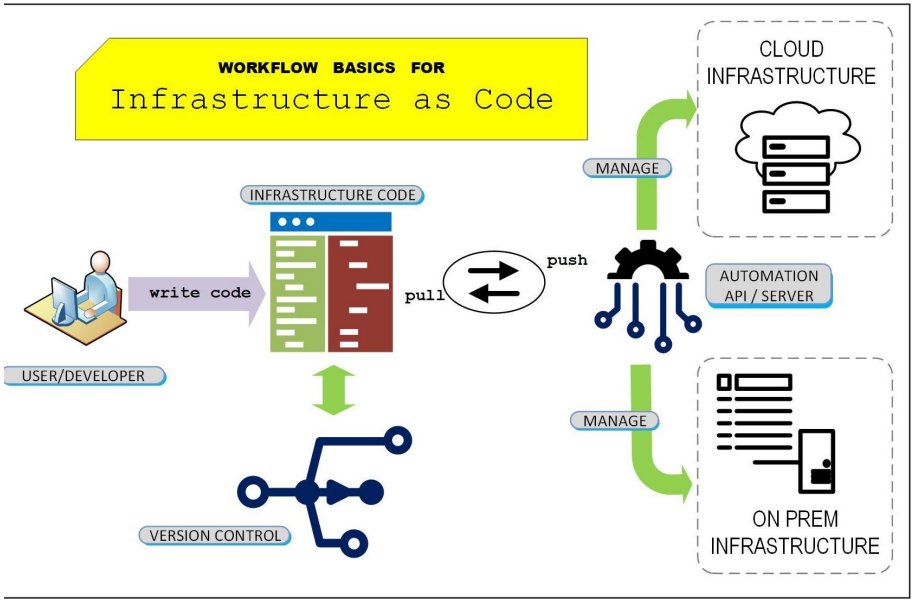
Terraform Full Course For Beginners

Click Here For Full Video Course (FREE)

What is Infrastructure as Code (IaC)?

A fundamental principle of DevOps is to treat infrastructure the same way developers treat code. Application code has a defined format and syntax. If the code is not written according to the rules of the programming language, applications cannot be created. Code is stored in a version management or source control system that logs a history of code development, changes, and bug fixes. When code is compiled or built into applications, we expect a consistent application to be created, and the build is repeatable and reliable. Example: Terraform, Chef, Puppet, Ansible etc





Infrastructure as Code (IaC) Benefits

Configuration consistency
IaC completely standardizes the setup of infrastructure so there is reduced possibility of any errors or deviations. This will decrease the chances of any incompatibility issues with your infrastructure and help your applications run more smoothly.

Cost reduction
By removing the manual component, people are able to refocus their efforts towards other tasks.

Speed
IaC allows faster execution when configuring infrastructure and aims at providing visibility to help other teams across the enterprise work quickly and more efficiently.

Stable and scalable environments
IaC delivers stable environments rapidly and at scale. Teams avoid manual configuration of environments and enforce consistency by representing the desired state of their environments via code. Infrastructure deployments with IaC are repeatable and prevent runtime issues caused by configuration drift or missing dependencies. DevOps teams can work together with a unified set of practices and tools to deliver applications and their supporting infrastructure rapidly, reliably, and at scale.

Reduced risk
Automation removes the risk associated with human error, like manual misconfiguration; removing this can decrease downtime and increase reliability.

Test
Infrastructure as Code enables DevOps teams to test applications in production-like environments early in the development cycle.

Accountability
Since you can version IaC configuration files like any source code file, you have full traceability of the changes each configuration suffered.

Documentation
Not only does IaC automate the process, but it also serves as a form of documentation of the proper way to instantiate infrastructure and insurance in the case where employees leave your company with institutional knowledge. Because code can be version-controlled, IaC allows every change to your server configuration to be documented, logged, and tracked. And these configurations can be tested, just like code.

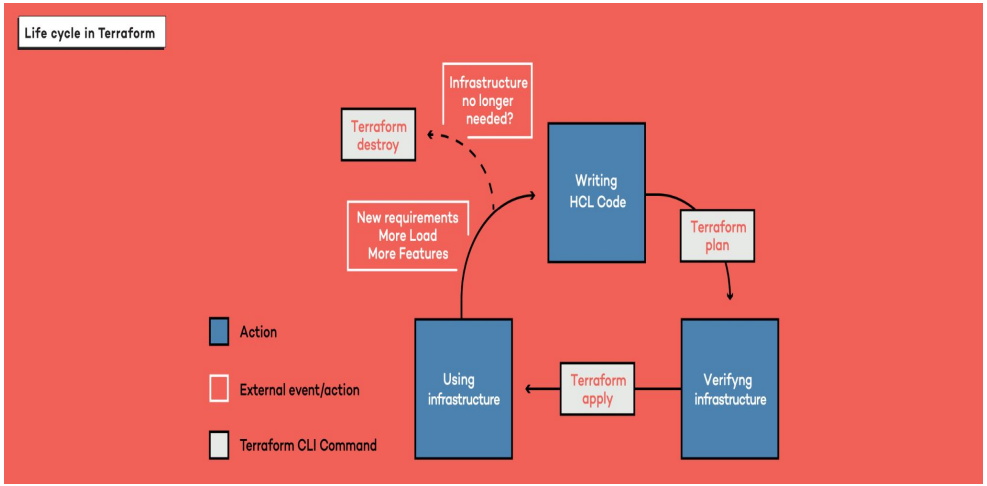
Enhanced security
If all compute, storage, and networking services are provisioned with code, then they are deployed the same way every time. This means that security standards can be easily and consistently deployed across company without having to have a security gatekeeper review and approve every change.

What is Terraform?

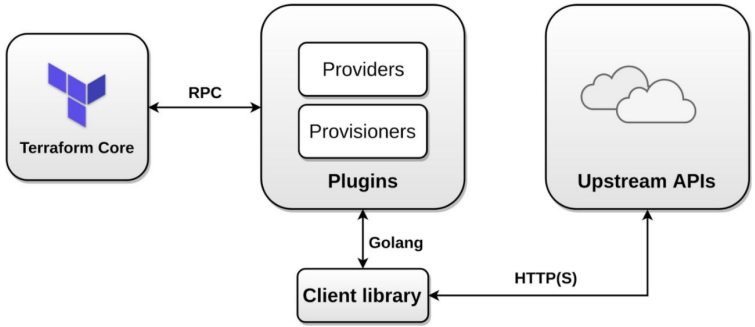
Terraform is an infrastructure as code (IaC) tool that allows you to build, change, and version infrastructure safely and efficiently. This includes low-level components such as compute instances, storage, and networking, as well as high-level components such as DNS entries, SaaS features, etc. Terraform can manage both existing service providers and custom in-house solutions.

Terraform has become one of the most popular **Infrastructure-as-code (IaC) tool,** getting used by DevOps team worldwide to automate infrastructure provisioning and management.

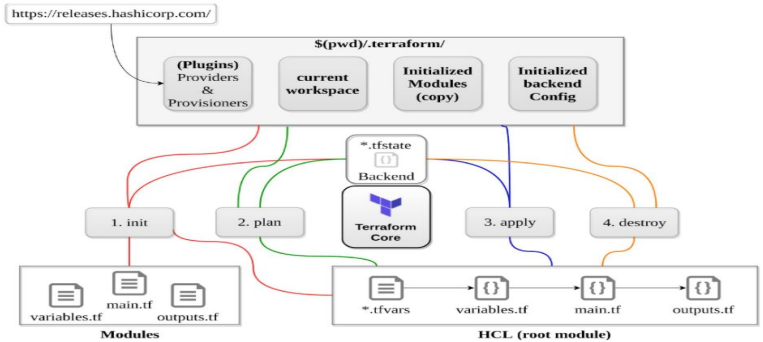
Terraform is an open-source, cloud-agnostic provisioning tool developed by HashiCorp and written in GO language.



Terraform Architecture



Terraform Workflow



Terraform Features

1) Improve multi-cloud infrastructure deployment
Terraform applies to multi-cloud scenarios, where similar infrastructure is deployed on Alibaba Cloud, other cloud providers, or local data centers. Developers can use the same tool and configuration file to simultaneously manage the resources of different cloud providers.

2) Automated infrastructure management
Terraform can create configuration file templates to define, provision, and configure ECS resources in a repeatable and predictable manner, reducing deployment and management errors resulting from human intervention. In addition, Terraform can deploy the same template multiple times to create the same development, test, and production environment.

3) Infrastructure as code
With Terraform, you can use code to manage and maintain resources. It allows you to store the infrastructure status, so that you can track the changes in different components of the system (infrastructure as code) and share these configurations with others.

4) Reduced development costs
You can reduce costs by creating on-demand development and deployment environments. In addition, you can evaluate such environments before making system changes.

5) Reduced time to provision
Traditional click-ops methods of deployment used by organizations can take days or even weeks, in addition to being error-prone. With Terraform, full deployment can take just minutes. For example, you can provision multiple Alibaba Cloud services at a time in a standardized way. Both brand new deployments and migrations can be done quickly and efficiently.

Terraform Basic Commands

After Installing Terraform CLI, if you type “terraform” in command line, you will see like below:

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

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

All other commands:
  console       Try Terraform expressions at an interactive command prompt
  fmt           Reformat your configuration in the standard style
  force-unlock  Release a stuck lock on the current workspace
  get           Install or upgrade remote Terraform modules
  graph         Generate a Graphviz graph of the steps in an operation
  import        Associate existing infrastructure with a Terraform resource
  login         Obtain and save credentials for a remote host
  logout        Remove locally-stored credentials for a remote host
  output        Show output values from your root module
  providers     Show the providers required for this configuration
  refresh       Update the state to match remote systems
  show          Show the current state or a saved plan
  state         Advanced state management
  taint         Mark a resource instance as not fully functional
  untaint       Remove the 'tainted' state from a resource instance
  version       Show the current Terraform version
  workspace     Workspace management

Global options (use these before the subcommand, if any):
  -chdir=DIR    Switch to a different working directory before executing the
                given subcommand.
  -help         Show this help output, or the help for a specified subcommand.
  -version      An alias for the "version" subcommand.
```

Among all these commands, we will be mostly using commands like:
validate, fmt, plan, apply , graph

validate: In simple terms, it’s check all the terraform files and confirm all files are proper , executable/deployable by terraform apply command

fmt: Just beautify the codes

plan: This will tell you what resources going to get affected and how.

apply: It will make all changes required to match the desired state of resources, but always run plan command before apply command

graph: It visually show all the resource relations

Terraform Get

The `terraform get` command is used to download and update modules mentioned in the root module.

Important points:

1) The modules are downloaded into a .terraform subdirectory of the current working directory. Don't commit this directory to your version control repository.
2)
3) The get command supports the following option:

- -update - If specified, modules that are already downloaded will be checked for updates and the updates will be downloaded if present.
- -no-color - Disable text coloring in the output.

terraform get	**downloads and update modules mentioned in the root module**
terraform get -update=true	**modules already downloaded will be checked for updates and updated**

Terraform Init

This command performs several different initialization steps in order to prepare the current working directory for use with Terraform.

Important points:

1) This command is always safe to run multiple times, to bring the working directory up to date with changes in the configuration.
2) Though subsequent runs may give errors, this command will never delete your existing configuration or state.
3) It initializes a working directory containing Terraform configuration files.
4) Performs backend initialization , storage for terraform state file, modules installation,
5) It download from terraform registry to local path, provider(s) plugins installation, the plugins are downloaded in the sub-directory of the present working directory at the path of .terraform/plugins
6) Supports -upgrade to update all previously installed plugins to the newest version that complies with the configuration version constraints,does not delete the existing configuration or state

terraform init	**initialize directory, pull down providers**
terraform init -get-plugins=false	**initialize directory, do not download plugins**
terraform init -verify-plugins=false	**initialize directory, do not verify plugins for Hashicorp signature**
terraform init -input=true	**Ask for input if necessary**
terraform init -lock=false	**Disable locking of state files during state-related operations**

Terraform Validate

This command check whether the execution plan for a configuration matches your expectations before provisioning or changing infrastructure.
The terraform validate command validates the configuration files in a directory, referring only to the configuration and not accessing any remote services such as remote state, provider APIs, etc.
Validate runs checks that verify whether a configuration is syntactically valid and internally consistent, regardless of any provided variables or existing state. It is thus primarily useful for general verification of reusable modules, including correctness of attribute names and value types.

Important points:

1) It validates syntactically for format and correctness.
2) It is used to validate/check the syntax of the Terraform files.
3) It verifies whether a configuration is syntactically valid and internally consistent, regardless of any provided variables or existing state.
4) A syntax check is done on all the terraform files in the directory, and will display an error if any of the files doesn't validate.

terraform validate	**validate configuration files for syntax**
terraform validate -backend=false	**validate code skip backend validation**

Terraform Plan

Run terraform plan to check whether the execution plan for a configuration matches your expectations before provisioning or changing infrastructure.
The terraform plan command creates an execution plan. By default, creating a plan consists of:

- Reading the current state of any already-existing remote objects to make sure that the Terraform state is up-to-date.
- Comparing the current configuration to the prior state and noting any differences.
- Proposing a set of change actions that should, if applied, make the remote objects match the configuration.

Important points:

1) It create a execution plan
2) It traverses each vertex and requests each provider using parallelism
3) It calculates the difference between the last-known state and
   the current state and presents this difference as the output of the terraform plan operation to user in their terminal
4) It does not modify the infrastructure or state.
5) It allows a user to see which actions Terraform will perform prior to making any changes to reach the desired state
6) It will scan all *.tf files in the directory and create the plan
7) It will perform refresh for each resource and might hit rate limiting issues as it calls provider APIs
8) All resources refresh can be disabled or avoided using
   a) -refresh=false or
   b) target=xxxx or
   c) break resources into different directories.
9) It supports -out to save the plan

terraform plan	**Creates an execution plan (dry run)**
terraform plan -out=path	**save generated plan output as a file**
terraform plan -destroy	**Outputs a destroy plan**

Terraform Apply

The terraform apply command executes the actions proposed in a Terraform plan.

The most straightforward way to use terraform apply is to run it without any arguments at all, in which case it will automatically create a new execution plan (as if you had run terraform plan) and then prompt you to approve that plan, before taking the indicated actions.

Another way to use terraform apply is to pass it the filename of a saved plan file you created earlier with terraform plan -out=..., in which case Terraform will apply the changes in the plan without any confirmation prompt. This two-step workflow is primarily intended for when running **Terraform in automation**.

Important Points

- It apply changes to reach the desired state.
- It scans the current directory for the configuration and applies the changes appropriately.
- It can be provided with a explicit plan, saved as out from terraform plan
- If no explicit plan file is given on the command line, terraform apply will create a new plan automatically and prompt for approval to apply it
- It will modify the infrastructure and the state.
- If a resource successfully creates but fails during provisioning,
  - Terraform will error and mark the resource as “tainted”.
  - A resource that is tainted has been physically created, but can’t be considered safe to use since provisioning failed.
  - Terraform also does not automatically roll back and destroy the resource during the apply when the failure happens, because that would go against the execution plan: the execution plan would’ve said a resource will be created, but does not say it will ever be deleted.
- It does not import any resource.
- It supports -auto-approve to apply the changes without asking for a confirmation
- It supports -target to apply a specific module

terraform apply	**Executes changes to the actual environment**
terraform apply –auto-approve	**Apply changes without being prompted to enter ”yes”**
terraform apply -refresh=true	**Update the state for each resource prior to planning and applying**
terraform apply -refresh=false	**do not reconcile state file with real-world resources(helpful with large complex deployments for saving deployment time)**
terraform apply -input=false	**Ask for input for variables if not directly set**
terraform apply -var ‘foo=bar’	**Set a variable in the Terraform configuration, can be used multiple times**
terraform apply -var-file=foo	**Specify a file that contains key/value pairs for variable values**
terraform apply -target	**Only apply/deploy changes to the targeted resource**
terraform apply plan.out	**use the plan.out plan file to deploy infrastructure**
terraform apply --parallelism=5	**number of simultaneous resource operations**
terraform apply -lock=true	**lock the state file so it can’t be modified by any other Terraform apply or modification action(possible only where backend allows locking)**

Terraform State

Terraform uses state data to remember which real-world object corresponds to each resource in the configuration; this allows it to modify an existing object when its resource declaration changes.

Terraform updates state automatically during plans and applies. However, it's sometimes necessary to make deliberate adjustments to Terraform's state data, usually to compensate for changes to the configuration or the real managed infrastructure.

Important Points:

- State helps keep track of the infrastructure Terraform manages
- It maps real world-world resources to terraform configuration
- It stored locally in the **terraform.tfstate** or can be stored in remote server such as Terraform Cloud, HashiCorp Consul, Amazon S3, Azure Blob Storage, Google Cloud Storage, Alibaba Cloud OSS
- It is recommended not to edit the state manually

terraform state list	**List all resources in the state file**
terraform state list resource_reference	**Only list resource with the given name**
terraform state mv	**Move an item in the state file**
terraform state rm	**Remove items from the state file**
terraform state pull	**Pull current state and output to stdout**
terraform state push	**Update remote state from a local state file**
terraform state show resource_reference	**Show the attributes of a single resource**
terraform refresh (Depreciated)	**reads the current settings from all managed remote objects and updates the Terraform state to match.**This won't modify your real remote objects, but it will modify the the Terraform state

resource_reference e.g. aws_instance.my_ec2_instance

Terraform Workspaces

In Terraform CLI, *workspaces* are separate instances of state data that can be used from the same working directory. You can use workspaces to manage multiple non-overlapping groups of resources with the same configuration.

Important Points

- It helps manage multiple distinct sets of infrastructure resources or environments with the same code.
- It just need to create needed workspace and use them, instead of creating a directory for each environment to manage
- state files for each workspace are stored in the directory terraform.tfstate.d
- terraform workspace new dev creates a new workspace and switches to it as well
- terraform workspace select dev helps select workspace
- terraform workspace list lists the workspaces and shows the current active one with *
- does not provide strong separation as it uses the same backend

terraform workspace new	**Create a new workspace and select it**
terraform workspace select	**Select an existing workspace**
terraform workspace list	**List the existing workspaces**
terraform workspace show	**Show the name of the current workspace**
terraform workspace delete	**Delete an empty workspace**

Example:

```
terraform workspace new dev
terraform workspace new test
terraform workspace new prod
terraform workspace select dev
terraform workspace select default
terraform workspace select prod
terraform workspace list
terraform workspace show
terraform workspace delete dev
```

Terraform Destroy

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

While you will typically not want to destroy long-lived objects in a production environment, Terraform is sometimes used to manage ephemeral infrastructure for development purposes, in which case you can use terraform destroy to conveniently clean up all of those temporary objects once you are finished with your work.

Important Points

- It destroy the infrastructure and all resources
- It modifies both state and infrastructure
- terraform destroy -target can be used to destroy targeted resources
- terraform plan -destroy allows creation of destroy plan

terraform destroy	**Destroy all remote objects managed by a particular Terraform configuration**
terraform destroy –auto-approve	**Destroy/cleanup without being prompted to enter ”yes”**
terraform destroy -target	**Only destroy the targeted resource and its dependencies**

Terraform Import

The terraform import command is used to import existing resources into Terraform.

Usage: terraform import [options] ADDRESS ID
Import will find the existing resource from ID and import it into your Terraform state at the given ADDRESS.

ADDRESS must be a valid resource address. Because any resource address is valid, the import command can import resources into modules as well as directly into the root of your state.
.

Important Points

- It helps import already-existing external resources, not managed by Terraform, into Terraform state and allow it to manage those resources
- Terraform is not able to auto-generate configurations for those imported modules, for now, and requires you to first write the resource definition in Terraform and then import this resource

terraform import aws_instance.foo i-abcd1234	**import an AWS instance with ID i-abcd1234 into aws_instance resource named "foo"**
terraform import module.foo.aws_instance.bar i-abcd1234	**import an AWS instance into the aws_instance resource named bar into a module named foo**

For More examples click here

Terraform Taint

The terraform taint command informs Terraform that a particular object has become degraded or damaged. Terraform represents this by marking the object as "tainted" in the Terraform state, in which case Terraform will propose to replace it in the next plan you create.

Note: It's depreciated , now it's recommended to use -replace option with terraform apply e.g.
terraform apply -replace="aws_instance.example[0]"

Important Points

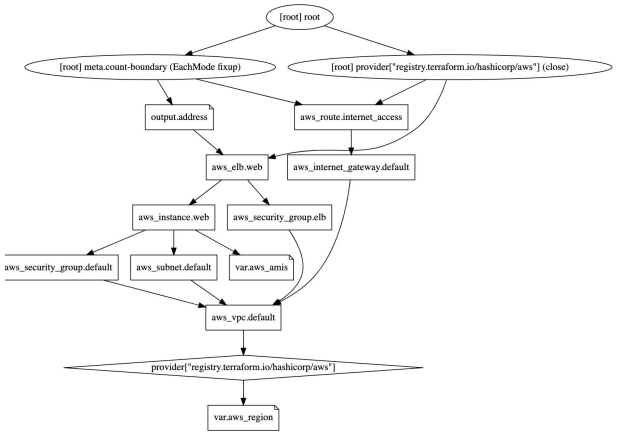
- It marks a Terraform-managed resource as tainted, forcing it to be destroyed and recreated on the next apply.
- It will not modify infrastructure, but does modify the state file in order to mark a resource as tainted. Infrastructure and state are changed in next apply.
- It can be used to taint a resource within a module

terraform taint aws_instance.my_ec2	**marks a Terraform-managed resource as tainted, forcing it to be destroyed and recreated on the next apply**
terraform untaint aws_instance.my_ec2	**unmarks a Terraform-managed resource as tainted**

Terraform Graph

The terraform graph command is used to generate a visual representation of either a configuration or execution plan. The output is in the DOT format, which can be used by GraphViz to generate charts.

The graph is outputted in DOT format. The typical program that can read this format is GraphViz, but many web services are also available to read this format.



`terraform graph | dot -Tsvg > graph.svg`

terraform graph creates a resource graph listing all resources in your configuration and their dependencies.

Terraform Coding

main.tf, variables.tf, outputs.tf. These are the recommended filenames to create infrastructure as code using terraform, main.tf should be the primary entrypoint.

Typically resource creation may be split into multiple files but any nested module calls should be in the main file. variables.tf and outputs.tf should contain the declarations for variables and outputs, respectively

HashiCorp Configuration Language (HCL)
Terraform projects are created using HashiCorp Configuration Language (HCL). HCL is a language derived from JSON that is optimised for efficiency, readability and simplicity. HCL only has a few conceptual keywords that can be used to describe infrastructures regardless of the desired target platform

Terraform version:
To check terraform version we can run command: terraform -version
In terraform main file i.e. main.tf e.g.

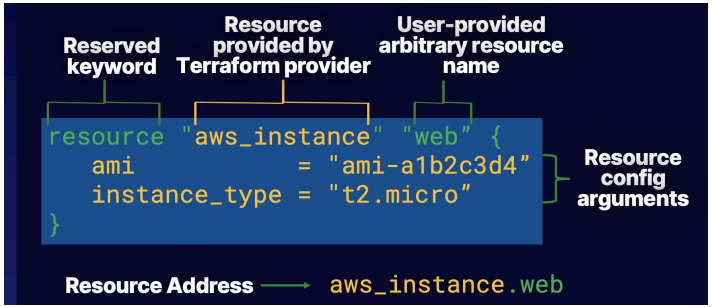
```
terraform {
 required_version = ">= 0.12"
}
```

Resource Declaration:

```
resource "aws_vpc" "main" {
  cidr_block = var.base_cidr_block
}
```

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

“resource” = Reserved Keyword
“aws_vpc” = Resource provide by Terraform Provider
“main” = User provided arbitrary resource name



- Resource are the most important element in the Terraform language that describes one or more infrastructure objects, such as compute instances etc

- Resource type and local name together serve as an identifier for a given resource and must be unique within a module *for e.g.* aws_instance.local_name

Terraform Coding

Data Sources

Data sources allow Terraform use information defined outside of Terraform, defined by another separate Terraform configuration, or modified by functions.

In addition to resources, data sources are another important component of Terraform projects. Data sources can be used to read values from existing services in the target platform.

E.g.

```
data "aws_ami" "example" {
  most_recent = true

  owners = ["self"]
  tags = {
    Name   = "app-server"
    Tested = "true"
  }
}
```

And then we can use the data source by referencing as data.aws_ami.example.id, more practical example:

```
data "aws_ami" "web" {
  filter {
    name   = "state"
    values = ["available"]
  }

  filter {
    name   = "tag:Component"
    values = ["web"]
  }

  most_recent = true
    }
```

Can be referenced as

```
resource "aws_instance" "web" {
  ami           = data.aws_ami.web.id
  instance_type = "t1.micro"
    }
```

Terraform Coding

Variables

Terraform projects can be made parameterisable with variables. Variables can have a default value. They are referenced in the project via their name.
The Terraform language includes a few kinds of blocks for requesting or publishing named values.

- Input Variables serve as parameters for a Terraform module, so users can customize behavior without editing the source.
- Output Values are like return values for a Terraform module.
- Local Values are a convenience feature for assigning a short name to an expression.

Input Variable

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

```
variable "image_id" {
  type        = string
  description = "The id of the machine image (AMI) to use for the server."
    }
```

Within the module that declared a variable, its value can be accessed from within expressions as var.<NAME>, where <NAME> matches the label given in the declaration block:

```
resource "aws_instance" "example" {
  instance_type = "t2.micro"
  ami           = var.image_id
    }
```

Types

The type argument in a variable block allows you to restrict the type of value that will be accepted as the value for a variable. If no type constraint is set then a value of any type is accepted.
The supported type keywords are:

- string
- number
- bool

The type constructors allow you to specify complex types such as collections:

- list(<TYPE>): a sequence of values identified by consecutive whole numbers starting with zero
- set(<TYPE>): a collection of values where each is identified by a string label.
- map(<TYPE>): a collection of unique values that do not have any secondary identifiers or ordering.
- object({<ATTR NAME> = <TYPE>, ... }): a collection of named attributes that each have their own type
- tuple([<TYPE>, ...]): a sequence of elements identified by consecutive whole numbers starting with zero, where each element has its own type

Terraform Coding

Ways to pass variables

Specify variable via argument

If only a few variables are defined in the project, or if existing values have to be explicitly overwritten, it is advisable to specify them as an argument of the corresponding Terraform command.

```
terraform apply -var="image_id=ami-abc123"
terraform apply -var='image_id_list=["ami-abc123","ami-def456"]'
-var="instance_type=t2.micro"
    terraform apply

    -var='image_id_map={"us-east-1":"ami-abc123","us-east-2":"ami-def456"}'
```

Variable Definition Precedence

Terraform loads variables in the following order, with later sources taking precedence over earlier ones:

- Environment variables
- The terraform.tfvars file, if present.
- The terraform.tfvars.json file, if present.
- Any *.auto.tfvars or *.auto.tfvars.json files, processed in lexical order of their filenames.
- Any -var and -var-file options on the command line, in the order they are provided. (This includes variables set by a Terraform Cloud workspace.)

Variable with .tfvars file

To set lots of variables, it is more convenient to specify their values in a *variable definitions file* (with a filename ending in either .tfvars or .tfvars.json) and then specify that file on the command line with -var-file:

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

In short: If a large number of variables have to be specified, an explicit .tfvars file should be created

A variable definitions file uses the same basic syntax as Terraform language files, but consists only of variable name assignments:

```
image_id = "ami-abc123"
availability_zone_names = [
  "us-east-1a",
  "us-west-1c",
]
```

Via environment Variable

Values for variables can also be defined as environment variables. The name of the Terraform variable has to be provided with the prefix TF_VAR_ .

E.g: `export TF_VAR_image_id=ami-abc123`

Terraform Coding

Variables input validation, custom validation rules

In addition to Type Constraints as described last slide, a module author can specify arbitrary custom validation rules for a particular variable using a validation block nested within the corresponding variable block:

```
variable "image_id" {
  type        = string
  description = "The id of the machine image (AMI) to use for the server."

  validation {
    condition     = length(var.image_id) > 4 && substr(var.image_id, 0, 4) == "ami-"
    error_message = "The image_id value must be a valid AMI id, starting with \"ami-\"."
  }
     }
```

Sensitive variables:

Setting a variable as sensitive prevents Terraform from showing its value in the plan or apply output, when you use that variable elsewhere in your configuration.

Terraform will still record sensitive values in the state, and so anyone who can access the state data will have access to the sensitive values in cleartext. For more information, see *Sensitive Data in State*.

Declare a variable as sensitive by setting the sensitive argument to true:

```
variable "user_information" {
  type = object({
    name    = string
    address = string
  })
  sensitive = true
}
resource "some_resource" "a" {
  name    = var.user_information.name
  address = var.user_information.address
}
```

String Interpolation

String interpolation is an integral part of the HCL.
Variables can be used via $ {} in strings
E.g.

```
variable "resource_prefix" {
type = string default = "example"
}

resource "azurerm_resource_group" "rg_demo" {
name = "${var.resource_prefix}_rg_demo"
location = var.target_location
}
```

Terraform Coding

Output Values

Output values are like the return values of a Terraform module, and have several uses:

- A child module can use outputs to expose a subset of its resource attributes to a parent module.
- A root module can use outputs to print certain values in the CLI output after running terraform apply.
- When using remote state, root module outputs can be accessed by other configurations via a terraform_remote_state data source

Important points:

- are like function return values.
- output can be marked as containing sensitive material using the optional sensitive argument, which prevents Terraform from showing its value in the list of outputs. However, they are still stored in the state as plain text.
- In a parent module, outputs of child modules are available in expressions as module.<MODULE NAME>.<OUTPUT NAME>.

E.g.

```
output "instance_ip_addr" {
  value       = aws_instance.server.private_ip
  description = "The private IP address of the main server instance."
    }
```

An output can be marked as containing sensitive material using the optional sensitive argument:

```
output "db_password" {
  value       = aws_db_instance.db.password
  description = "The password for logging in to the database."
  sensitive   = true
    }
```

Terraform Coding

Local Values

A local value assigns a name to an expression, so you can use it multiple times within a module without repeating it.
In short: Local values are like a function's temporary local variables.

Important Points:

- locals assigns a name to an expression, allowing it to be used multiple times within a module without repeating it.
- are like a function's temporary local variables.
- helps to avoid repeating the same values or expressions multiple times in a configuration.

Example:

```
locals {
 service_name = "forum"
 owner       = "Community Team"
    }
```

Or

```
locals {

 # Ids for multiple sets of EC2 instances, merged together
 instance_ids = concat(aws_instance.blue.*.id, aws_instance.green.*.id)
}

locals {
 # Common tags to be assigned to all resources
 common_tags = {
   Service = local.service_name
   Owner   = local.owner
 }
    }
```

Use like below:

```
resource "aws_instance" "example" {
 # ...

 tags = local.common_tags
}
```

Terraform Coding

Providers

Terraform relies on plugins called "providers" to interact with cloud providers, SaaS providers, and other APIs.
Terraform configurations must declare which providers they require so that Terraform can install and use them. Additionally, some providers require configuration (like endpoint URLs or cloud regions) before they can be used.

As per Terraform, **Provider** is responsible for understanding API interactions and exposing resources.

Of Course It support: Alibaba Cloud, AWS, GCP, Microsoft Azure, here these are treated as A Provider.
Here is the details list of Providers (Providers get added / updated time to time):

ACME, Akamai, Alibaba Cloud, Archive, Arukas, Avi Vantage, Aviatrix, AWS, Azure, Azure Active Directory, Azure Stack, A10, Networks, Bitbucket, Brightbox, CenturyLinkCloud, Chef, CherryServers, Circonus, Cisco ASA, Cisco ACI, Cloudflare,CloudScale.ch, CloudStack, Cobbler, Consul, Datadog, DigitalOcean, DNS, DNSimple, DNSMadeEasy, Docker, , ,Dome9 ,Dyn,Exoscale ,External,F5 BIG-IP,Fastly,FlexibleEngine, FortiOS, Genymotion, GitHub, GitLab, Google Cloud Platform, Grafana, Gridscale, Hedvig, Helm, Heroku, Hetzner Cloud, HTTP, HuaweiCloud, HuaweiCloudStack, Icinga2, Ignition, InfluxDB, JDCloud, Kubernetes, LaunchDarkly, Librato, Linode, Local, Logentries, LogicMonitor, Mailgun, MongoDB Atlas, MySQL, Naver Cloud, Netlify, New Relic, Nomad, NS1, Null, Nutanix, 1&1, OpenNebula, OpenStack, OpenTelekomCloud, OpsGenie, Oracle Cloud Infrastructure, Oracle Cloud Platform, Oracle Public Cloud, OVH, Packet, PagerDuty, Palo Alto Networks, PostgreSQL, PowerDNS, ProfitBricks, Pureport, RabbitMQ, RancherRancher2Random, , RightScale, Rundeck, RunScope, Scaleway, Selectel, SignalFx, Skytap, SoftLayer, Spotinst, StatusCake, TelefonicaOpenCloud, Template, TencentCloud, Terraform, Terraform Cloud, TLS, Triton, UCloud, UltraDNS, Vault, Venafi, VMware NSX-T, VMware vCloud Director, VMware vRA7, VMware vSphere, Vultr, Yandex

E.g.

```
provider "aws" {
 region = var.aws_region
}
```

Then use like:

```
# Create a VPC to launch our instances into
resource "aws_vpc" "default" {
 cidr_block = "10.0.0.0/16"
}
```

Terraform Coding

In build Functions

The Terraform language includes a number of built-in functions that you can call from within expressions to transform and combine values. The general syntax for function calls is a function name followed by comma-separated arguments in parentheses:

```
max(5, 12, 9)
```

Below are the types of function with function names:

Numeric Functions : abs, ceil, floor, log, max, min, parseint, pow, signum

String Functions: chomp, format, formalist, join, lower, regex, regexall, rep[lace, split, strrev, title, trim, trimprefix, trimsuffix, trimspace, upper

Collection Functions: alltrue, anytrue, chunklist, coalesce, coalescelist, compact, concat, contains, distinct, element, flatten, index, keys, length, list, lookup, map, matchkeys,merge, one, range, reverse, setintersection, setproduct, setsubsctract, setunion, slice, sort, sum, transpose, values, zipmap

Encoding Functions: base643ncode, base64decode, base64gzip, csvdecode, jsonencode, jsondecode, urlencode, yamlencode, yamldecode

Filesystem functions: absath, dirname, pathexpand, basename, file, fileexists, fileset, filebase64, templatefile

Date & Time Functions: formade, timeadd, timestamp

Hash and Crypto Functions : base64sha256, base64sha512, bcrypt, filebase64sha512, filemd5, filesha1, filesha256, filesha512, md5, rsadecrypt, sha, sha256, sha512, uuid, uuidv5

IP Network Functions: cidrhost, cidrnetmask, cidrsubnets

Type Conversion functions: can, defaults, nonsensative, sensitive, tobool, tolist, tomap, tonumber, toset, tostring, try

Terraform Coding

Modules

A Terraform module is a set of Terraform configuration files in a single directory. Even a simple configuration consisting of a single directory with one or more .tf files is a module. When you run Terraform commands directly from such a directory, it is considered the root module. So in this sense, every Terraform configuration is part of a module. You may have a simple set of Terraform configuration files such as:

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.

With modules, reusable components can be created. Each Terraform project is a valid Terraform module. In addition, modules can be created in sub-folders within a project. Variables serve as the configuration interface of modules. For each module to be used, a module block must be defined in the project.

Root Module:
Every Terraform configuration has at least one module, known as its *root module*, which consists of the resources defined in the .tf files in the main working directory.

Child Modules
A Terraform module (usually the root module of a configuration) can *call* other modules to include their resources into the configuration. A module that has been called by another module is often referred to as a *child module.*

Child modules can be called multiple times within the same configuration, and multiple configurations can use the same child module.

Root module example

```
.
├── LICENSE
├── README.md
├── main.tf
├── variables.tf
└── outputs.tf
```

Child Modules

```
├── README.md
├── main.tf
├── variables.tf
├── outputs.tf
├── ...
├── modules/
│   ├── nestedA/
│   │   ├── README.md
│   │   ├── variables.tf
│   │   ├── main.tf
│   │   └── outputs.tf
│   ├── nestedB/
│   ├── .../
├── examples/
│   ├── exampleA/
│   │   ├── main.tf
│   ├── exampleB/
│   ├── .../
```

Terraform Coding

Using Modules

A module can call other modules, which lets you include the child module's resources into the configuration in a concise way. Modules can also be called multiple times, either within the same configuration or in separate configurations, allowing resource configurations to be packaged and re-used.

Calling a child module:

To *call* a module means to include the contents of that module into the configuration with specific values for its input variables. Modules are called from within other modules using module blocks:

```
module "servers" {
  source = "./app-cluster"

  servers = 5
    }
```

The source argument is mandatory for all modules.

Accessing Module Output Values:

The resources defined in a module are encapsulated, so the calling module cannot access their attributes directly. However, the child module can declare output values to selectively export certain values to be accessed by the calling module.

For example, if the ./app-cluster module referenced in the example above exported an output value named instance_ids then the calling module can reference that result using the expression module.servers.instance_ids:

```
resource "aws_elb" "example" {
  # ...

  instances = module.servers.instance_ids
}
```

Using version:

When using modules installed from a module registry, it's recommend explicitly constraining the acceptable version numbers to avoid unexpected or unwanted changes.

Use the version argument in the module block to specify versions:

```
module "consul" {
  source  = "hashicorp/consul/aws"
  version = "0.0.5"
    }
```

The source argument in a module block tells Terraform where to find the source code for the desired child module.

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

Terraform Coding

Provisioners:

Provisioners can be used to model specific actions on the local machine or on a remote machine in order to prepare servers or other infrastructure objects for service.
Terraform includes the concept of provisioners as a measure of pragmatism, knowing that there will always be certain behaviors that can't be directly represented in Terraform's declarative model.
Important points:

- It run code locally or remotely on resource creation
  - local exec executes code on the machine running terraform
  - remote exec
    - runs on the provisioned resource
    - supports ssh and winrm
  - requires inline list of commands
- It should be used as a last resort
- This are defined within the resource block.
- It support types – Create and Destroy
  - if creation time fails, resource is tainted if provisioning failed, by default. (next apply it will be re-created)
  - behavior can be overridden by setting the on_failure to continue, which means ignore and continue
  - for destroy, if it fails – resources are not removed

Example: Local

```
resource "aws_instance" "web" {
  # ...

  provisioner "local-exec" {
    command = "echo The server's IP address is
${self.private_ip}"
  }
    }
```

Example: Remote

```
resource "aws_instance" "web" {
  # ...

  provisioner "remote-exec" {
    inline = [
      "sudo apt-get -y update",
      "sudo apt-get -y install nginx",
      "sudo service nginx start",
    ]

  }
    }
```

Terraform Coding

Life Cycle Properly / meta argument:

The behaviour of Terraform in the context of a resource can be influenced with meta arguments. The lifecycle property offers three important levers to individualise the Terraform life cycle of a resource. E.g.

```
resource "aws_instance" "example" {
 # ...

 lifecycle {
   create_before_destroy = true
 }
    }
```

Ignore Manual Changes

If a resource is manipulated with Terraform after it has been created, these adjustments would be overwritten by executing terraform apply again. This behaviour can be deactivated. e.g.

```
resource "aws_instance" "example" {
 # ...

 lifecycle {
   ignore_changes = [
     # Ignore changes to tags, e.g. because a management agent
     # updates these based on some ruleset managed elsewhere.
     tags,
   ]
 }
    }
```

Prevent deletion

The destruction of a resource with Terraform can be prevented by the property prevent_destroy of the lifecycle block:

```
resource "aws_instance" "example" {
 # ...
 lifecycle {
  Prevent_destroy = true
 }
    }
```

Sequence Control for Adjustments

Changing some properties makes it necessary to destroy and recreate a resource (for example changing the name of an Azure VM). With create_before_destroy you can influence the chronological order:

```
resource "aws_instance" "example" {
 # ...

 lifecycle {
   create_before_destroy = true
 }
    }
```

Terraform Coding

Remote State Backends

If Terraform projects are supervised by several people or are used by systems such as CI/CD servers, the status (Terraform State) must be kept centrally. This ensures that no congruent writing operations are in progress and that each execution can use the latest state. So-called remote state backends allow the Terraform State to be consumed by remote systems.

Terraform uses persistent state data to keep track of the resources it manages. Since it needs the state in order to know which real-world infrastructure objects correspond to the resources in a configuration, everyone working with a given collection of infrastructure resources must be able to access the same state data.

The local backend stores state as a local file on disk, but every other backend stores state in a remote service of some kind, which allows multiple people to access it. Accessing state in a remote service generally requires some kind of access credentials, since state data contains extremely sensitive information.

Some backends act like plain "**remote disks**" for state files; others support *locking* the state while operations are being performed, which helps prevent conflicts and inconsistencies.
Options are: artifactory, azurerm, consul, cos, etcd, etdcv3, gcs, http, kubernetes, manta, oss, pg, s3, swift

With AWS S3

Stores the state as a given key in a given bucket on Amazon S3. This backend also supports state locking and consistency checking via Dynamo DB, which can be enabled by setting the dynamodb_table field to an existing DynamoDB table name.

```
terraform {
  backend "s3" {
    bucket = "mybucket"
    key    = "path/to/my/key"
    region = "us-east-1"
  }
    }
```

With GCP

```
terraform {
  backend "gcs" {
    bucket  = "tf-state-prod"
    prefix  = "terraform/state"
  }
    }
```

With Azure

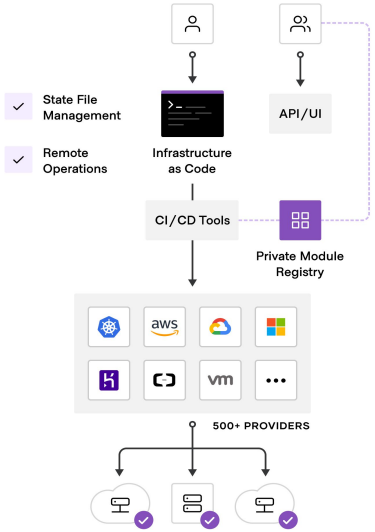
Stores the state as a Blob with the given Key within the Blob Container within the Blob Storage Account. This backend also supports state locking and consistency checking via native capabilities of Azure Blob Storage.

```
terraform {
  backend "azurerm" {
    resource_group_name  = "StorageAccount-ResourceGroup"
    storage_account_name = "abcd1234"
    container_name       = "tfstate"
    key                  = "prod.terraform.tfstate"
  }
    }
```

Terraform Cloud

Terraform Cloud is HashiCorp's managed service offering that eliminates the need for unnecessary tooling and documentation to use Terraform in production.

Provision infrastructure securely and reliably in the cloud with free remote state storage. As you scale, add workspaces for better collaboration with your team.



Terraform Cloud features

Remote Terraform Execution
Supports Remote Operations for Remote Terraform execution which helps provide consistency and visibility for critical provisioning operations.

Remote State Management
acts as a remote backend for the Terraform state. State storage is tied to workspaces, which helps keep state associated with the configuration that created it.

Workspaces
organizes infrastructure with workspaces instead of directories. Each workspace contains everything necessary to manage a given collection of infrastructure, and Terraform uses that content whenever it executes in the context of that workspace.

Version Control Integration
is designed to work directly with the version control system (VCS) provider.

Private Module Registry
provides a private and central library of versioned & validated modules to be used within the organization

Team based Permission System – can define groups of users that match the organization's real-world teams and assign them only the permissions they need

Sentinel Policies
embeds the Sentinel policy-as-code framework, which lets you define and enforce granular policies for how the organization provisions infrastructure. Helps eliminate provisioned resources that don't follow security, compliance, or operational policies.

Cost Estimation
can display an estimate of its total cost, as well as any change in cost caused by the proposed updates

Security – encrypts state at rest and protects it with TLS in transit.

Command:

terraform login : obtain and save API token for Terraform cloud

terraform logout : Log out of Terraform Cloud, defaults to hostname app.terraform.io

Remote backend:

```
terraform {
  backend "remote" {
    hostname = "app.terraform.io"
    organization = "my-org"
    workspaces {
      name = "my-workspace"
    }
  }
}
```

Terraform Enterprise

Terraform Enterprise is self-hosted distribution of Terraform Cloud. It offers enterprises a private instance of the Terraform Cloud application, with no resource limits and with additional enterprise-grade architectural features like audit logging and SAML single sign-on.

Important points:

- It includes all the Terraform Cloud features as described in previous slide
- It supports detailed audit logging and tracks the identity of the user requesting state and maintains a history of state changes.
- SAML for SSO provides the ability to govern user access to your applications.

A Terraform Enterprise install that is provisioned on a network that does not have Internet access is generally known as an **air-gapped install**. These types of installs require you to pull updates, providers, etc. from external sources vs. being able to download them directly.

Terraform Enterprise currently supports running under the following operating systems for a Clustered deployment:

- Ubuntu 16.04.3 – 16.04.5 / 18.04
- Red Hat Enterprise Linux 7.4 through 7.7
- CentOS 7.4 – 7.7
- Amazon Linux
- Oracle Linux
- Clusters currently don't support other Linux variants

Good luck!

I hope you'll use this knowledge and build awesome solutions.

Watch the FULL **video course here** (**FREE**):

If any issue contact me in Linkedin:
https://www.linkedin.com/in/sandip-das-developer/