

DevOps
Cloud
Computing

Caltech

Center for Technology &
Management Education

Post Graduate Program in DevOps

DevOps
Cloud
Computing



Caltech

**Center for Technology &
Management Education**

Configuration Management with Ansible and Terraform



Terraform Loops, Built-in Functions, and Provisioners

A Day in the Life of a DevOps Engineer

Your organization has recently started using Terraform for infrastructure automation.

However, the DevOps team is facing trouble making complex terraform code. They are looking for a solution that can help make complex code more readable, concise, and understandable by encapsulating the repetitive code in a single block.

As a Lead-DevOps Engineer, you have to demonstrate several built-in functions that can be called within expressions to transform and combine values.



A Day in the Life of a DevOps Engineer

The team is also looking for an option that can be used to model certain operations on a local or remote system to get servers or other infrastructure objects ready to use.

Last but not least, you will need to explain the Terraform workflow to them so that they can easily write, plan and provision reproducible infrastructure.

To achieve all of the above, along with some additional features, you will be learning a few concepts in this lesson that will help find a solution for the given scenario.



Learning Objectives

By the end of this lesson, you will be able to:

- Understand how Terraform code can be made dynamic for reusability using loops
- Call Built-in Functions within expressions to transform and combine values
- Model certain operations on a local or remote system
- Use the unique self-object



Terraform Loops

Terraform Loops

Loops are widely used in the Terraform community to make modules dynamic.

- Loops can help make complex code more readable, concise, and understandable by encapsulating the repetitive code in a single block.
- It can make infrastructure scalable and efficient with just a few lines of code.
- Components and lists can be iterated to build looping logic, which is both human-readable and machine-friendly.

Example

If multiple discs or IP addresses are required, a Virtual Machine module can use loops to deploy them.

Terraform Loops

Terraform provides a variety of looping constructions, each designed for a specific scenario:

01

count parameter: loop over resources

02

for_each expressions: loop over resources and inline blocks within a resource

03

for expressions: loop over lists and maps

Terraform Loops: Count Parameter

Every Terraform resource has a meta-parameter called ***count*** that defines the number of resources to be created.

Example

```
resource "aws_iam_user" "example"  
{  
  count = 3  
  name  = "neo"  
}
```

In this example, three IAM users are created. It uses the **aws_iam_user** resource to create an IAM user.

However, the above example will create all three IAM users with the same name, which would result in an error because usernames are required to be unique.

Terraform Loops: Count Parameter

A user can use ***count.index*** to get the index of each iteration in the loop to provide each user a unique name.

Example

```
resource "aws_iam_user"  
  "example" {  
    count = 3  
    name  = "neo.${count.index}"  
  }  
}
```

When a user runs the plan command on the preceding code, they will notice that Terraform wants to create three IAM users, each with a slightly different name ("neo.0," "neo.1," "neo.2").

A user can further personalize each iteration of the loop by combining ***count.index*** with Terraform built-in functions.

Terraform Loops: Count Parameter

Here's an example of a complete Terraform file with the count parameter included.

```
provider "aws" {
  region      = "eu-central-1"
  access_key  = "AKIATQ37NXB2OBQHAALW"
  secret_key  = "ilKygurap8zSErv7jySTDi2796WGqMkEtN6txxxx"
}

resource "aws_instance" "ec2_example" {
  ami          = "ami-0767046d1677be5a0"
  instance_type = "t2.micro"
  count = 1

  tags = {
    Name = "Terraform EC2"
  }
}

resource "aws_iam_user" "example" {
  count = length(var.user_names)
  name  = var.user_names[count.index]
}

variable "user_names" {
  description = "IAM usernames"
  type        = list(string)
  default     = ["user1", "user2", "user3"]
}
```

Once the user applies this terraform configuration using the terraform apply command, it will perform the following actions on AWS-

- Create one ec2 instance.
- Create three IAM users - user1, user2, user3.

Terraform Loops: For_each Expressions

The `for_each` expression can be used to create multiple copies of a complete resource, or multiple copies of an inline-block within a resource by looping over lists, sets, and maps.

Example

```
resource "aws_iam_user" "example"
{
  for_each = toset(var.user_names)
  name     = each.value
}
```

This is an example of creating three IAM users using the **for_each** expression.

In the above code, the usage of **toset** is to convert the `var.user_names` list into a set because `for_each` only supports sets and maps when used on a resource.

Terraform Loops: For_each Expressions

Here is the complete terraform file with the implementation of for_each.

```
provider "aws" {
  region      = "eu-central-1"
  access_key  = "AKIATQ37NXB2NN3D4ARS"
  secret_key  = "3v9mlwZQvmccL3ouldxiDeEf1bWaG3kccpVlXXXX"
}

resource "aws_instance" "ec2_example" {
  ami          = "ami-0767046d1677be5a0"
  instance_type = "t2.micro"
  count = 1

  tags = {
    Name = "Terraform EC2"
  }
}

resource "aws_iam_user" "example" {
  for_each = var.user_names
  name     = each.value
}

variable "user_names" {
  description = "IAM usernames"
  type        = set(string)
  default     = ["user1", "user2"]
}
```

terraform
m apply

```
# aws_iam_user.example["user1"] will be created
+ resource "aws_iam_user" "example" {
  + arn          = (known after apply)
  + force_destroy = false
  + id           = (known after apply)
  + name         = "user1"
  + path         = "/"
  + tags_all     = (known after apply)
  + unique_id    = (known after apply)
}

# aws_iam_user.example["user2"] will be created
+ resource "aws_iam_user" "example" {
  + arn          = (known after apply)
  + force_destroy = false
  + id           = (known after apply)
  + name         = "user2"
  + path         = "/"
  + tags_all     = (known after apply)
  + unique_id    = (known after apply)
}
```


Terraform Loops: For Expressions

For expressions in loops can be used to generate a single value.

The basic syntax of a **for** expression is:

```
[for <ITEM> in <LIST> : <OUTPUT>]
```

In this syntax:

- LIST is a list to loop over.
- ITEM is the local variable name to assign to each item in the LIST.
- OUTPUT is an expression that transforms ITEM in some way.

Terraform Loops: For Expressions

Terraform code to convert the list of names in var.names to uppercase:

Example

```
variable "names" {  
  description = "A list of names"  
  type        = list(string)  
  default     = ["neo", "trinity", "morpheus"]  
}  
output "upper_names" {  
  value = [for name in var.names : upper(name)]  
}
```

Terraform Loops: For Expressions

If the user runs terraform apply on this code, the output will be as follows:

Example

```
$ terraform apply

Apply complete! Resources: 0 added, 0 changed, 0 destroyed.

Outputs:

upper_names = [
  "NEO",
  "TRINITY",
  "MORPHEUS",
]
```

A user can filter the resulting list by defining a condition, just like with Python's list comprehensions.

Conditionals

There are multiple ways to do conditionals, just as there are several ways to handle loops in Terraform. Each is designed to be used in a slightly different scenario:

01

count parameter: conditional resources

02

for_each and **for** expressions: conditional resources and inline blocks within a resource

Assisted Practice

Implementing Loops with Count

Duration: 10 Min.

Problem Statement:

You've been given the task of demonstrating Loops with Count for creating a resource as many times as the code specifies.

Assisted Practice: Guidelines

Steps to be followed:

1. Creating a main file
2. Executing the main file using loops and count



Terraform Built-in Functions

Built-in Functions

The Terraform language includes several built-in functions that can be called within expressions to transform and combine values.

A function name followed by comma-separated arguments in parentheses is the general syntax for function calls.

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

Terraform only supports built-in functions because user-defined functions are not supported by the Terraform language.

Built-in Functions

The following are some examples of built-in functions in Terraform:

`abs()`

Returns the absolute value of the given number

`chomp()`

Removes newline characters at the end of a string

`alltrue()`

Returns true if all elements in the given collection are true or "true". It also returns true if the collection is empty

`base64decode()`

Takes a string containing a base64 character sequence and returns the original string

Built-in Functions

`abspath()`

Converts a string containing a filesystem path to an absolute path

`formatdate()`

Converts a timestamp into a different time format

`md5()`

Computes the md5 hash of a given string and encodes it with hexadecimal digits

`cidrhost()`

Calculates a full host IP address for a given host number within the prefix of an IP network address

`can()`

Returns a boolean value indicating whether the provided expression delivered an error-free outcome

Built-in Functions

The types of Built-in Functions supported by the Terraform language are:

Type Conversion Functions

Numeric Functions

String Functions

Collection Functions

Encoding Functions

Filesystem Functions

Date and Time Functions

Hash and Crypto
Functions

IP Network Functions

Built-in Functions: Numeric

The following are some of the most used built-in numeric functions:

abs()	floor()
ceil()	log()
max()	min()
signum()	pow()
parseint()	

Built-in Functions: String

The following are some of the most used built-in string functions:

chomp()	split()
format()	strrev()
formatlist()	substr()
indent()	title()
join()	trim()
lower()	trimprefix()
regex()	trimsuffix()
regaxall()	trimspace()
replace()	upper()

Built-in Functions: Collection

The following are some of the most used built-in collection functions:

alltrue()	distinct()	map()	setsubtract()
anytrue()	element()	matchkeys()	setunion()
chunklist()	flatten()	merge()	slice()
coalesce()	index()	one()	sort()
coalescelist()	keys()	range()	sum()
compact()	length()	reverse()	transpose()
concat()	list()	setintersection()	values()
contains()	lookup()	setproduct()	zipmap()

Built-in Functions: Encoding

The following are some of the most used built-in encoding functions:

base64decode()	base64encode
textencodebase64()	textdecodebase64()
base64gzip()	jsonencode()
csvdecode()	urlencode()
jsondecode()	yamldecode()
yamlencode()	

Built-in Functions: Filesystem

The following are some of the most used built-in filesystem functions:

abspath()	file()
dirname()	fileexists()
pathexpand()	fileset()
basename()	filebase64()
templatefil()	

Built-in Functions: Date and Time, and IP Network

The following are some of the most used built-in date and time functions:

formatdate()	timeadd()
timestamp()	

The following are some of the most used built-in IP network functions:

cidrhost()	cidrsubnet()
cidrnetmask()	cidrsubnets()

Built-in Functions: Hash and Crypto Functions

The following are some of the most used built-in hash and crypto functions:

base64sha256()	base64sha512()
filesha512()	md5()
bcrypt()	rsadecrypt()
sha256()	sha1()
filebase64sha512()	filebase64sha256()
filemd5()	sha512()
filesha1()	uuid()
filesha256()	uuidv5()

Built-in Functions: Type Conversion Functions

The following are some of the most used built-in type conversion functions:

can	tolist
defaults	tomap
nonsensitive	tonumber
sensitive	toset
tobool	tostring
try()	

Assisted Practice

Using Terraform Dynamic Blocks and Built-in Functions to Deploy to AWS

Duration: 10 Min.

Problem Statement:

You've been requested to demonstrate how to deploy to AWS using Terraform Dynamic Blocks and Built-in Functions.

Assisted Practice: Guidelines

Steps to be followed:

1. Copying VPC ID
2. Creating a terraform main file
3. Executing the main file
4. Verifying rules creation



Terraform Provisioners

Provisioners

Provisioners can be used to model certain operations on a local or remote system to get servers or other infrastructure objects ready to use.

Terraform includes Provisioners as a measure of pragmatism, knowing that certain behaviors would never be directly represented in Terraform's declarative model.

Provisioners

Provisioners can be used to solve the following scenarios in general. However, Provisioners are not necessary as there is always a better alternative.

01

Passing data into virtual machines and other compute resources

02

Running configuration management software

Note

It is technically possible to run the CLI for the target system using the local-exec provisioner. It can be used to create, update, or interact with remote objects in a system.

How to Use Provisioners?

Users can place a provisioner block inside the resource block of a compute instance.

Example

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

- Except **local-exec** provisioner, other provisioners must connect to the remote system using SSH or WinRM.
- Terraform needs to know how to communicate with the server, thus a user must include a **connection** block.

Terraform includes both built-in and third-party provisioners. However, it is not recommended to use any provisioner except the built-in file, local-exec, and remote-exec provisioners.

Provisioners: The Self Object

Expressions in provisioner blocks cannot refer to their parent resource by name. Instead, they use the unique self object.

The self-object represents the provisioner's parent resource and has all the resource attributes.

Example

A user can use **self.public_ip** to reference an aws_instance's public_ip attribute.

Note

Values, such as sensitive variables, may be used in provisioner block settings. In this case, the provisioner's log output is automatically suppressed to avoid displaying sensitive values.

Creation and Destroy Time Provisioners

Creation-Time Provisioners

- Provisioners run by default when the resource they are defined within is created.
- The resource is marked as tainted if a creation-time provisioner fails. On the next terraform apply, a tainted resource will be planned for destruction and recreation.

Destroy-Time Provisioners

- If **when = destroy** is specified, the provisioner will run when the resource, it is defined within, is destroyed.
- Destroy provisioners are run before the resource is destroyed. If they fail, Terraform will throw an error and rerun the provisioners on the next terraform apply.

Multiple Provisioners

A resource can have multiple provisioners assigned to it. The configuration file specifies the order in which multiple provisioners should be executed.

Example of multiple provisioners:

```
resource "aws_instance" "web" {  
  # ...  
  
  provisioner "local-exec" {  
    command = "echo first"  
  }  
  
  provisioner "local-exec" {  
    command = "echo second"  
  }  
}
```

Failure Behavior

Provisioners that fail by default will also cause the terraform apply to fail. This can be changed by using the `on_failure` setting.

Example

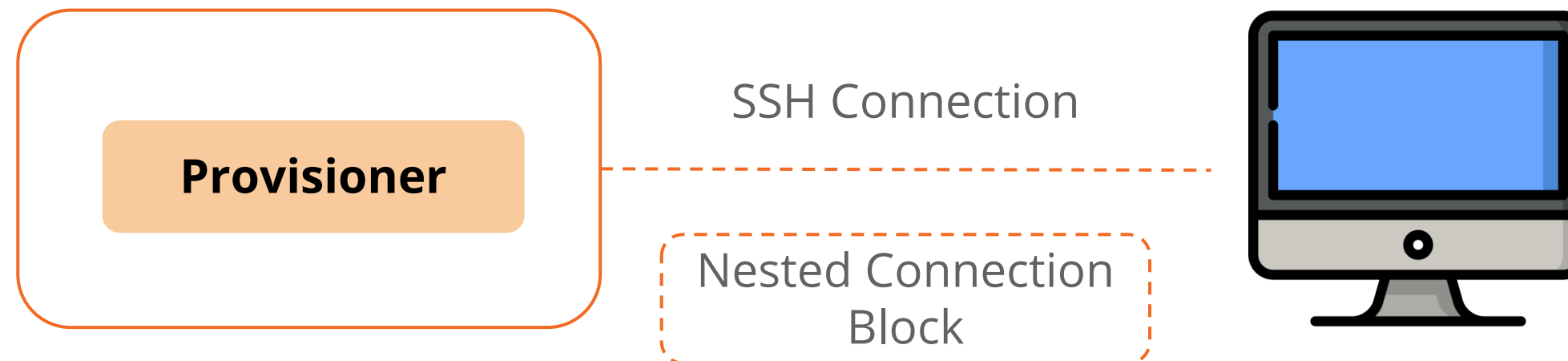
```
resource "aws_instance" "web" {  
  # ...  
  
  provisioner "local-exec" {  
    command      = "echo The server's IP address  
is ${self.private_ip}"  
    on_failure = continue  
  }  
}
```

The allowed values are:

- **Continue:** Ignore the error and continue to create or destroy
- **Fail:** Raise an error and stop applying (the default behavior). If this is a creation provisioner, taint the resource

Provisioner Connection

Most provisioners expect a nested connection block containing data about how to connect to the remote resource through SSH or WinRM.



Provisioner Connection

- Connection blocks don't require a block label and can be nested within a resource or a provisioner.
- All provisioners in a resource are affected by a connection block nested directly within it.
- A connection block that is nested inside a provisioner block affects only that provisioner and overrides any resource-level connection settings.

One use case for providing multiple connections is to have an initial provisioner connect as the root user to set up user accounts and have subsequent provisioners connect as a user with limited permissions.

Provisioner Connection

Provisioner Connection is demonstrated in the following example:

```
# Copies the file as the root user using SSH
provisioner "file" {
  source      = "conf/myapp.conf"
  destination = "/etc/myapp.conf"

  connection {
    type      = "ssh"
    user      = "root"
    password  = "${var.root_password}"
    host      = "${var.host}"
  }
}

# Copies the file as the Administrator user using WinRM
provisioner "file" {
  source      = "conf/myapp.conf"
  destination = "C:/App/myapp.conf"

  connection {
    type      = "winrm"
    user      = "Administrator"
    password  = "${var.admin_password}"
    host      = "${var.host}"
  }
}
```

Provisioner Connection: Self Object

Connection block expressions cannot refer to their parent resource by name. They can use the self object.

The self object represents the connection's parent resource and has all that resource's attributes.

Example

A user can use **self.public_ip** to reference an **aws_instance's public_ip** attribute.

Provisioner Connection: Arguments

All Connection types support the following arguments:

type

The type of connection that should be used

user

The user for the connection

password

The password for the connection

Provisioner Connection: Arguments

host

The address of the resource to connect to

port

The port to connect to

time-out

The timeout to wait for the connection to become available

script_path

The path used to copy scripts meant for remote execution

Provisioner Connection: Arguments

SSH connection type supports the following Arguments:

private_key

The contents of an SSH key to use for the connection

**certificate
agent**

The contents of a signed CA certificate

agent

Set to false to disable using ssh-agent to authenticate

Provisioner Connection: Arguments

agent_identity

The preferred identity from the SSH agent for authentication

host_key

The public key from the remote host or the signing CA to verify the connection

target_platform

The target platform to connect to

Note

If the platform is set to windows, the default script_path is **"c:\windows\temp\terraform_%RAND%.cmd"**, assuming the SSH default shell is **cmd.exe**.
If the SSH default shell is PowerShell, set script_path to **"c:/windows/temp/terraform_%RAND%.ps1"**.

Provisioner Connection: Arguments

WinRM connection type supports the following Arguments:

https

Set as true to connect using HTTPS instead of HTTP

insecure

Set as true to invalidate the HTTPS certificate chain

use_ntlm

Set as true to use NTLM authentication, rather than the default

cacert

The CA certificate to validate against the connection

Connecting Through a Bastion Host with SSH

The SSH connection also supports the following fields to facilitate connections via a bastion host.

bastion_host

This enables the bastion host connection

bastion_host_key

The public key from the remote host or the signing CA used to verify the host connection

bastion_port

The port used to connect to the bastion host

Connecting Through a Bastion Host with SSH

bastion_user

The user for the connection to the bastion host

bastion_password

The password used for the bastion host

bastion_private_key

The contents of an SSH key file to use for the bastion host

bastion_certificate

The contents of a signed CA Certificate

Provisioners Without a Resource

If a user needs to run provisioners that aren't directly associated with a specific resource, they can associate them with a **null_resource**.

```
resource "aws_instance" "cluster" {
  count = 3
  # ...
}
resource "null_resource" "cluster" {
  # Changes to any instance of the cluster requires re-provisioning
  triggers = {
    cluster_instance_ids = "${join(",", aws_instance.cluster.*.id)}"
  }
  # Bootstrap script can run on any instance of the cluster
  # So we just choose the first in this case
  connection {
    host = "${element(aws_instance.cluster.*.public_ip, 0)}"
  }
  provisioner "remote-exec" {
    # Bootstrap script called with private_ip of each node in the cluster
    inline = [
      "bootstrap-cluster.sh ${join(" ", aws_instance.cluster.*.private_ip)}",
    ]
  }
}
```

Null_resource supports the following argument:

triggers: A map of values that should cause this set of provisioners to rerun

Terraform Provisioners Types

Types of Terraform Provisioners

There are two types of Terraform Provisioners:

Generic Provisioner

- File
- Local-exec
- Remote-exec

Vendor Provisioner

- Chef
- Habitat
- Puppet
- Salt-masterclass

Generic Provisioner: File

The file provisioner copies files and directories from the Terraform-running system to the newly formed resource. Both SSH and WinRM connections can be used with the file provisioner.

File provisioner supports the following arguments:

source

The source file or folder

content

The content to copy on the destination

destination

The destination path

Note

A directory can also be uploaded to the remote machine using the file provisioner.

Generic Provisioner: Local-Exec

The local-exec provisioner calls a local executable after the resource creation.

The local-exec provisioner supports The following arguments

command

The command to execute

working_dir

Specifies the working directory where command will be executed

interpreter

A list of interpreter arguments used to execute the command

environment

Block of key-value pairs representing the environment of the executed command

Generic Provisioner: Remote-Exec

The remote-exec provisioner invokes a script on a remote resource after its creation.

The remote-exec provisioner supports the following arguments:

inline

A list of command strings

script

A path to a local script that will be copied to the remote resource and then executed

scripts

A list of paths to local scripts that will be copied to the remote resource and then executed

Vendor Provisioner: Chef

The Chef provisioner installs, configures, and runs the Chef client on a remote resource. Chef provisioner supports both SSH and WinRM connections.

The chef Provisioner has the following prerequisites for various connection types:

- Curl must be available on the remote host or SSH type connections.
- PowerShell 2.0 must be available on the remote host or WinRM connections,

The chef end-user license agreement must be accepted by setting chef_license to accept in client_options argument unless the user is installing an old version of the Chef client.

Vendor Provisioner: Chef

The chef provisioner supports the following arguments:

node_name (string)

The name of the node to register with the Chef server

server_url (string)

The URL to the Chef server

user_name (string)

The name of an existing Chef user to register the new Chef client and optionally configure Chef vaults

user_key (string)

The contents of the user key that will be used to authenticate with the Chef server

It also supports multiple other optional arguments, such as **attributes_json (string)** and **wait_for_retry (integer)**.

Vendor Provisioner: Habitat

The Habitat provisioner installs the Habitat supervisor and loads configured services. This provisioner only supports Linux targets using the SSH connection type currently.

The Habitat provisioner has some prerequisites for specific connection types:

A few tools to be available on the remote host for SSH type connections:

- curl
- tee
- setsid - Only if using the unmanaged service type

Habitat: Argument Reference

There are 2 configuration levels: supervisor and service.

- Supervisor configurations are placed directly within the provisioner block.
- A provisioner can define zero or more services to execute, each with its service block within the provisioner.
- A service block can also contain zero or more bind blocks to build service group bindings.

Supervisor Arguments

accept_license (bool): Set to true to accept Habitat end-user license agreement

Supervisor configuration also supports multiple other optional arguments, such as version (string) and builder_auth_token (string).

Habitat: Argument Reference

Service Arguments

name (string): The Habitat package identifier of the service to run (ie core/haproxy or core/redis/3.2.4/20171002182640)

Service configuration also supports multiple other optional arguments, such as binds (array) and service_key (string).

Vendor Provisioner: Puppet

The puppet provisioner installs, configures, and runs the Puppet agent on a remote resource. The puppet provisioner supports both SSH and WinRM type connections.

The puppet provisioner has some prerequisites for specific connection types:

- Curl must be available on the remote host for SSH type connections.
- PowerShell 2.0 must be available on the remote host or WinRM type connection.
- The puppet provisioner requires Bolt to be installed on a workstation with the following modules
 - danieldreier/autosign
 - puppetlabs/puppet_agent

Vendor Provisioner: Puppet

The puppet provisioner supports the following arguments:

server (string): The FQDN of the Puppet master that the agent is to connect to

It also supports multiple other optional arguments, such as **server_user (string)** and **bolt_timeout (string)**.

Vendor Provisioner: Salt Masterclass

The salt masterless Terraform provisioner uses salt states to provision Terraform machines without requiring a salt master. The salt masterless provisioner supports SSH connections.

The presence of **curl** on the remote host is required for salt masterless to work.

Argument Reference

The only required argument supported by the salt masterclass provisioner is the path to the user's local salt state tree.

It also supports multiple other optional arguments, such as **bootstrap_args (string)** and **salt_bin_dir (string)**.

Assisted Practice

Implementing Local-Exec Provisioners

Duration: 10 Min.

Problem Statement:

You've been tasked with demonstrating how to use Local-Exec Provisioner to run a local executable after a resource has been created.

Assisted Practice: Guidelines

Steps to be followed:

1. Downloading the appropriate package
2. Adding the binary file into the bin directory
3. Using local-exec provisioners
4. Initializing terraform
5. Running our deployment and passing our variable



Assisted Practice

Implementing Remote-Exec Provisioners

Duration: 10 Min.

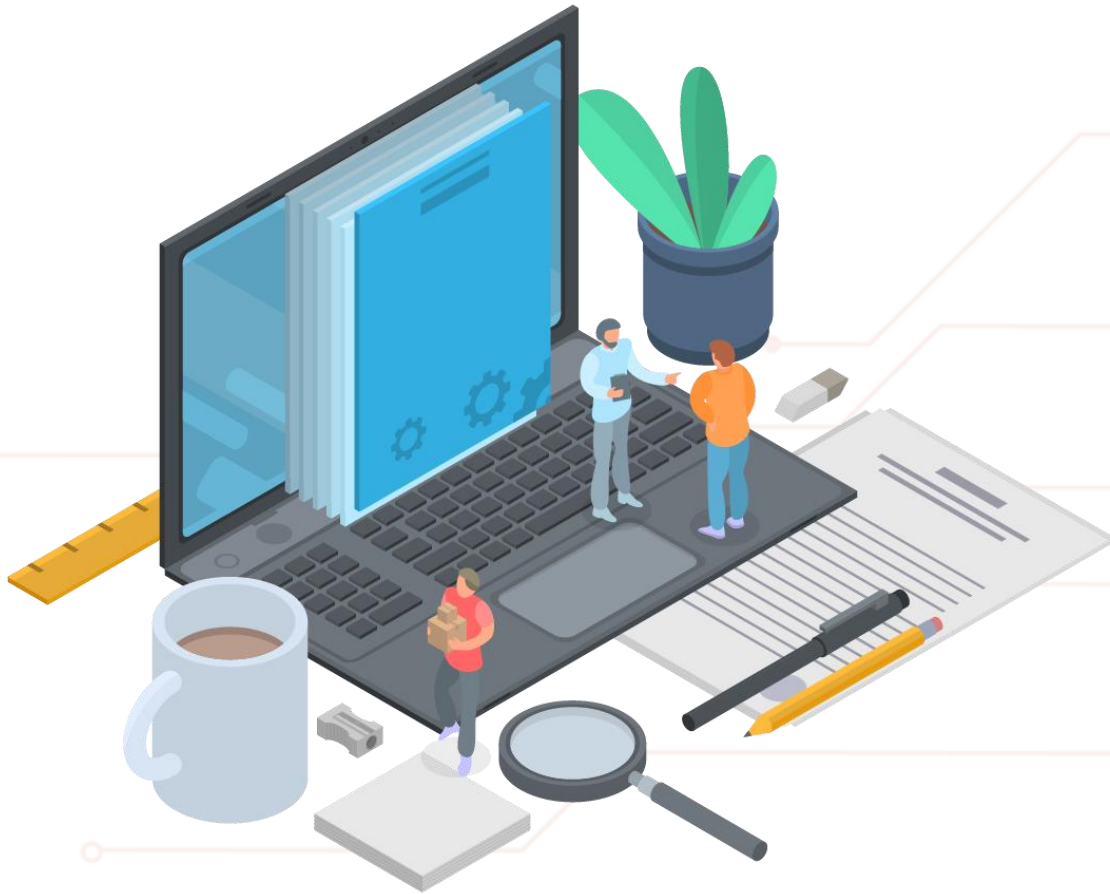
Problem Statement:

You've been tasked with demonstrating how to use Remote-Exec Provisioner to run a script on a remote resource after it's been created. The provisioner can be used to execute a configuration management tool, bootstrap into a cluster, and so on.

Assisted Practice: Guidelines

Steps to be followed:

1. Verifying the terraform installation
2. Using remote-exec provisioners
3. Initializing and deploying terraform



Terraform Workflow

Terraform Workflow

The steps of the Terraform Workflow are:

Write

Author
infrastructure as
code

Plan

Preview changes
before applying

Apply

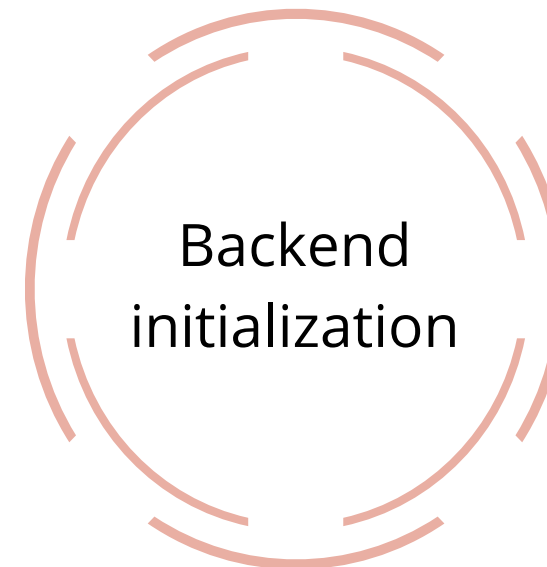
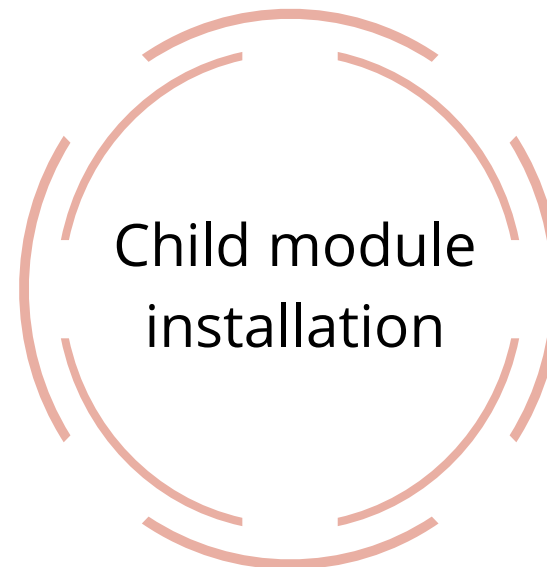
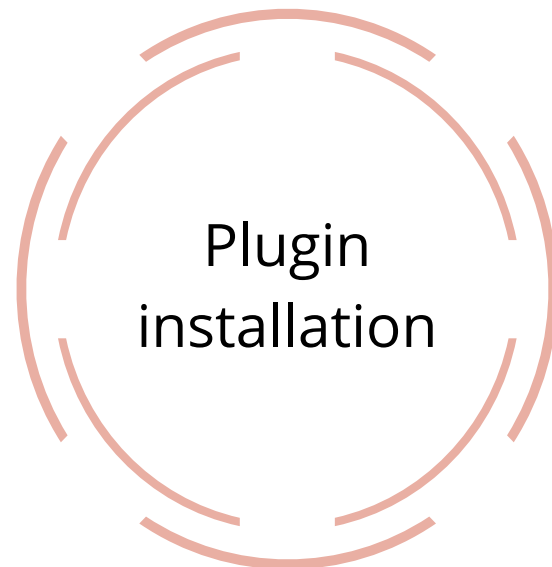
Provision
reproducible
infrastructure

Terraform Workflow

The code is initialized using the Terraform **init** command, after writing the code in Terraform.

This command initializes the working directory containing Terraform configuration files.

Init command can also be used for:



Terraform Workflow

The Terraform can be used in the following ways:



As an individual user



As a team



As an entire organization

Workflow: Individual User

Write



Users write Terraform configurations just like writing a code, in an editor of the user's choice.



Repeatedly running plans help flush out syntax errors and ensures that the config is coming together as expected.



This parallels working on application code as an individual, where a strict feedback loop between editing code and running test commands is useful.

Workflow: Individual User

Plan

Commit the work and review the final plan once the write step's feedback loop has produced a change.

```
$ git add main.tf
$ git commit -m 'Managing infrastructure as code!'

[main (root-commit) f735520] Managing infrastructure as code!
1 file changed, 1 insertion(+)
```

Terraform apply will display a plan for confirmation before changing any infrastructure.

Workflow: Individual User

Apply

The user can instruct Terraform to provision real infrastructure after one last check.

It's common to push version control repository to a remote location for safekeeping, at this point.

```
$ git remote add origin https://github.com/*user*/*repo*.git  
$ git push origin main
```

Workflow: Team

Once multiple people collaborate on Terraform configuration, new steps are added to each part of the core workflow to ensure everyone is working together smoothly.



www.shutterstock.com · 1214730637

Workflow: Team

Write

- While individuals of a team make changes to the Terraform configuration in the editor of choice, the changes are saved to version control branches to avoid colliding with each other's work.
- Working in branches enables team members to resolve mutually incompatible infrastructure changes using their normal merge conflict workflow.

```
$ git checkout -b add-load-balancer  
Switched to a new branch 'add-load-balancer'
```

Workflow: Team

Write

- As the team and the infrastructure grow, the number of sensitive input variables (API keys, SSL cert pairs) required to run a plan also grows.
- To avoid the burden and the security risk, each team member can arrange all sensitive inputs locally.
- It's common for teams to migrate to a model in which Terraform operations are executed in a shared Continuous Integration (CI) environment.

Workflow: Team

Plan

Terraforms plan output creates an opportunity for team members to review each other's work for the teams collaborating on infrastructure.

These reviews occur alongside pull requests within version control, the point at which an individual proposes a merge from their working branch to the shared team branch.

Some teams that run Terraform locally make a practice that pull requests should include an attached copy of the speculative plan output generated by the change author.

Workflow: Team

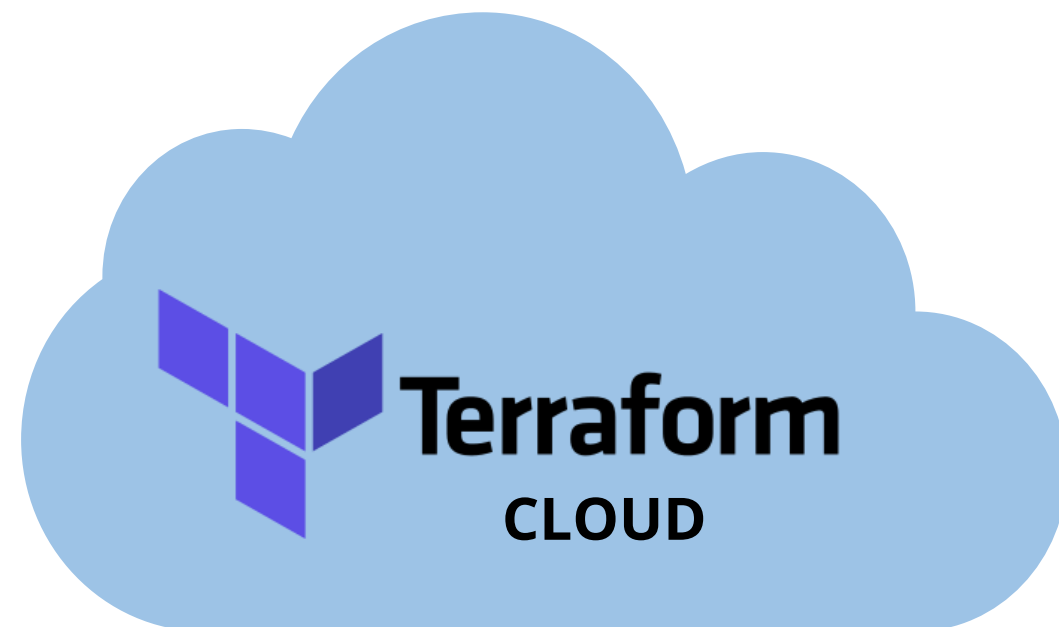
Apply

Once a pull request is approved and merged, the team reviews the final concrete plan that runs against the shared team branch and the latest version of the state file.

- This plan has the potential to be different than the one reviewed on the pull request.
- It is at this point that the team asks questions about the potential implications of applying the change.
- Depending on the change, sometimes team members check the apply output during execution.

Workflow: Terraform Cloud

While the individual and team workflows enable the safe, predictable, and reproducible creating or changing of infrastructure, multiple collaboration points can be streamlined, especially as teams and organizations scale.



Terraform Cloud supports and enhances the core Terraform workflow for anyone collaborating on infrastructure, from small teams to large organizations.

Workflow: Terraform Cloud

Terraform Cloud provides a centralized and secure location for storing input variables and state while bringing back a strict feedback loop for speculative plans for config authors.

Write

- Terraform configuration interacts with the Terraform Cloud via the "remote" backend.

```
terraform {  
  backend "remote" {  
    organization = "my-org"  
    workspaces {  
      prefix = "my-app-"  
    }  
  }  
}
```

Workflow: Terraform Cloud

Write

- Terraform Cloud API key helps team members to edit config and run speculative plans against the latest version of the state file.
- Team members can work on the authoring config until it is ready to be proposed as a change via a pull request, with the assistance of plan output.

Workflow: Terraform Cloud

Terraform Cloud makes the process of reviewing a speculative plan easier for team members.

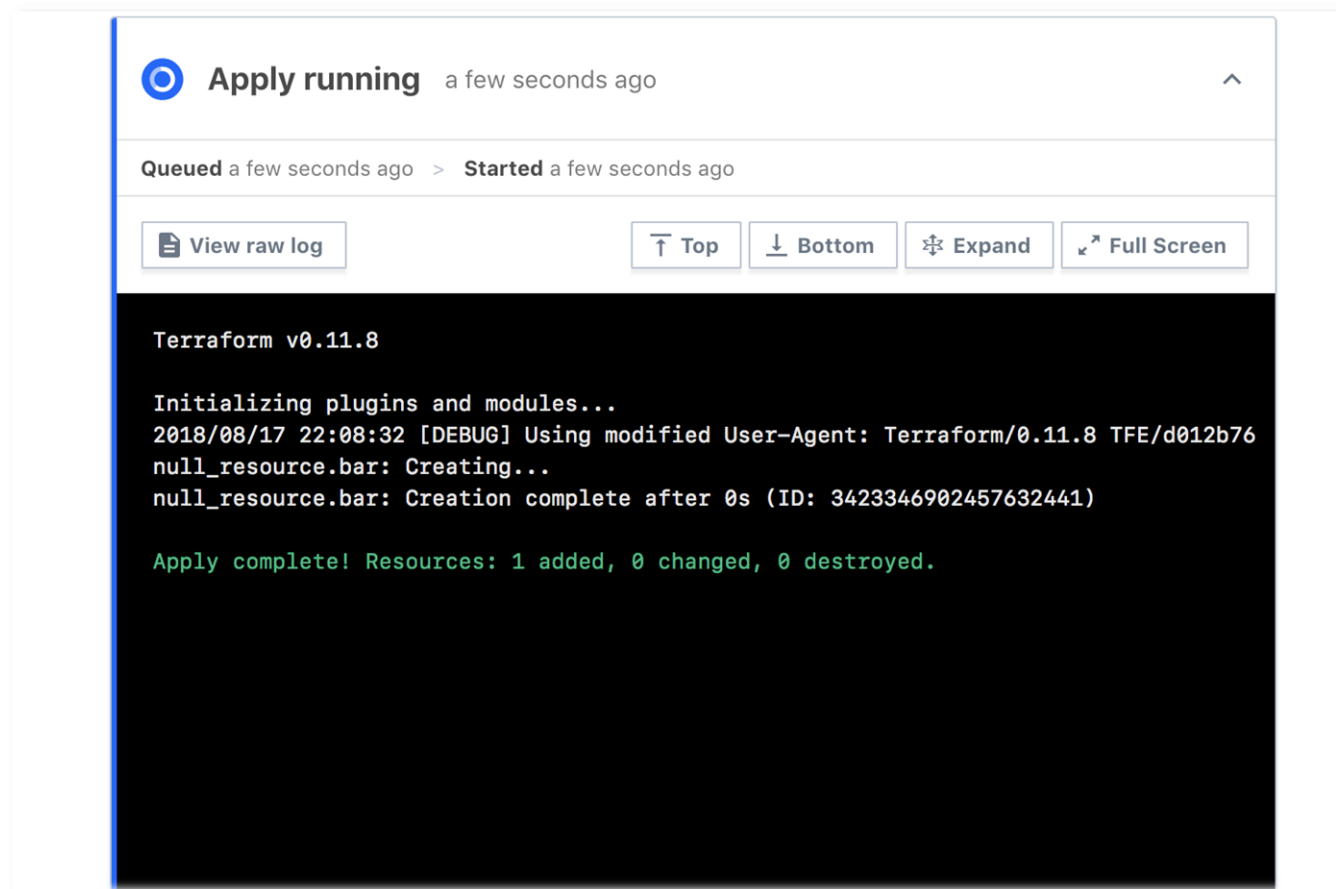
Plan

- The plan automatically runs when the pull request is created.
- Status updates to the pull request indicate while the plan is in progress.
- Once the plan is completed, the status update indicates the changes in the speculative plan from the pull request view.

Workflow: Terraform Cloud

Apply

After the merge, Terraform Cloud presents the concrete plan to the team for review and approval.



The screenshot shows the Terraform Cloud console interface. At the top, it says "Apply running" with a circular progress indicator and "a few seconds ago". Below this, it shows the status "Queued a few seconds ago" and "Started a few seconds ago". There are buttons for "View raw log", "Top", "Bottom", "Expand", and "Full Screen". The main area displays the Terraform v0.11.8 logs, which include the following text:

```
Terraform v0.11.8
Initializing plugins and modules...
2018/08/17 22:08:32 [DEBUG] Using modified User-Agent: Terraform/0.11.8 TFE/d012b76
null_resource.bar: Creating...
null_resource.bar: Creation complete after 0s (ID: 3423346902457632441)

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

The team can discuss any outstanding questions about the plan before the change is made.

Key Takeaways

- Loops are widely used in the Terraform community to make modules dynamic.
- The Terraform language includes several built-in functions that can be called within expressions.
- Provisioners can be used to model certain operations on a local or remote system.
- Provisioners expect a nested connection block containing data about how to connect to the remote resource through SSH or WinRM.



Build Infrastructure

Duration: 25 Min.

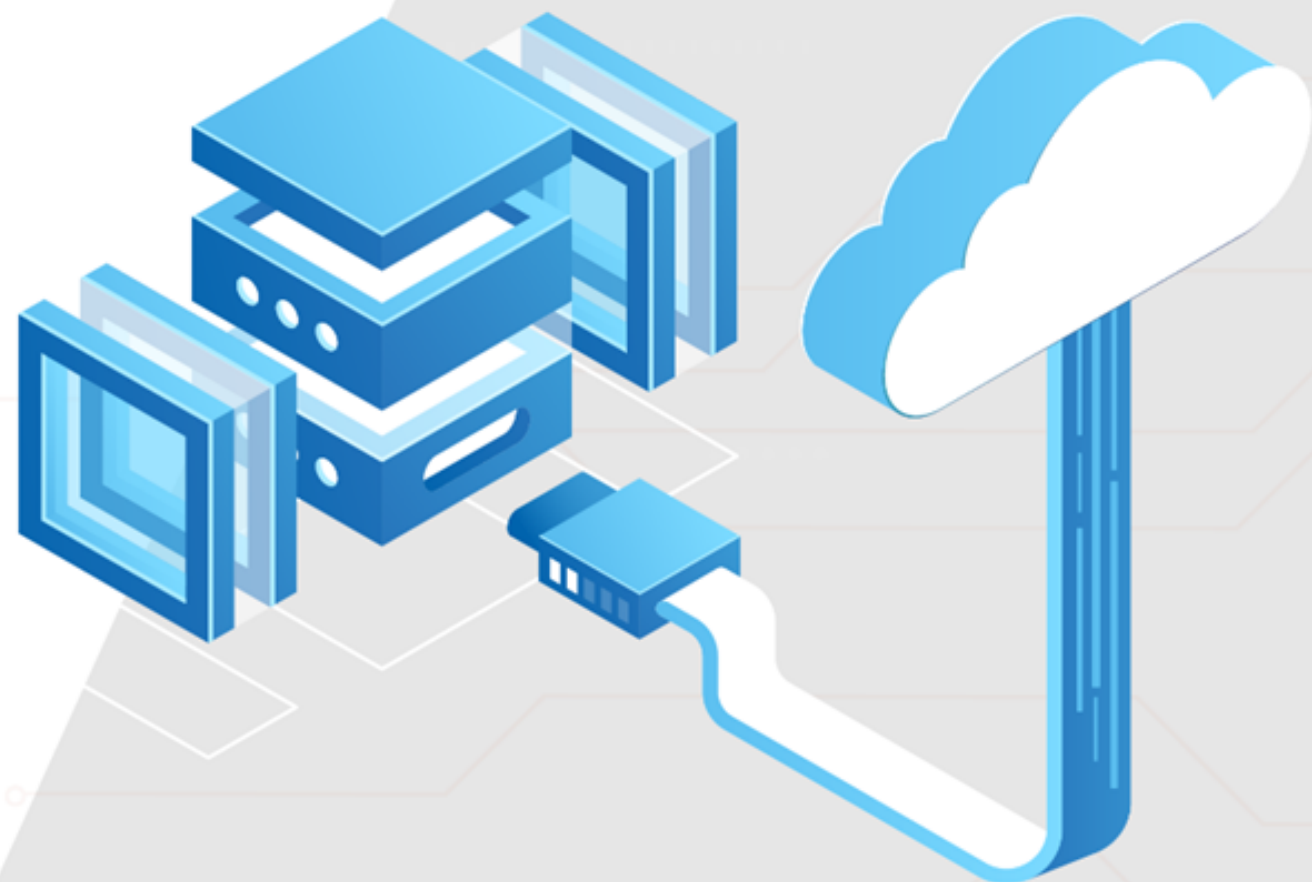


Project agenda: To build an AWS Infrastructure

Description: Terraform is an infrastructure as code (IaC) tool that allows you to build, change, and version infrastructure safely and efficiently. You can declare infrastructure components in configuration files that are then used by Terraform to provision, adjust, and tear down infrastructure in various cloud providers. In this project, you will provision an EC2 instance on Amazon Web Services (AWS). EC2 instances are virtual machines running on AWS and are a common component of many infrastructure projects.

Perform the following:

- Configuring the AWS CLI from the terminal
- Formatting and validating the configuration
- Creating the infrastructure
- Validating the creation of EC2 instance



Thank you