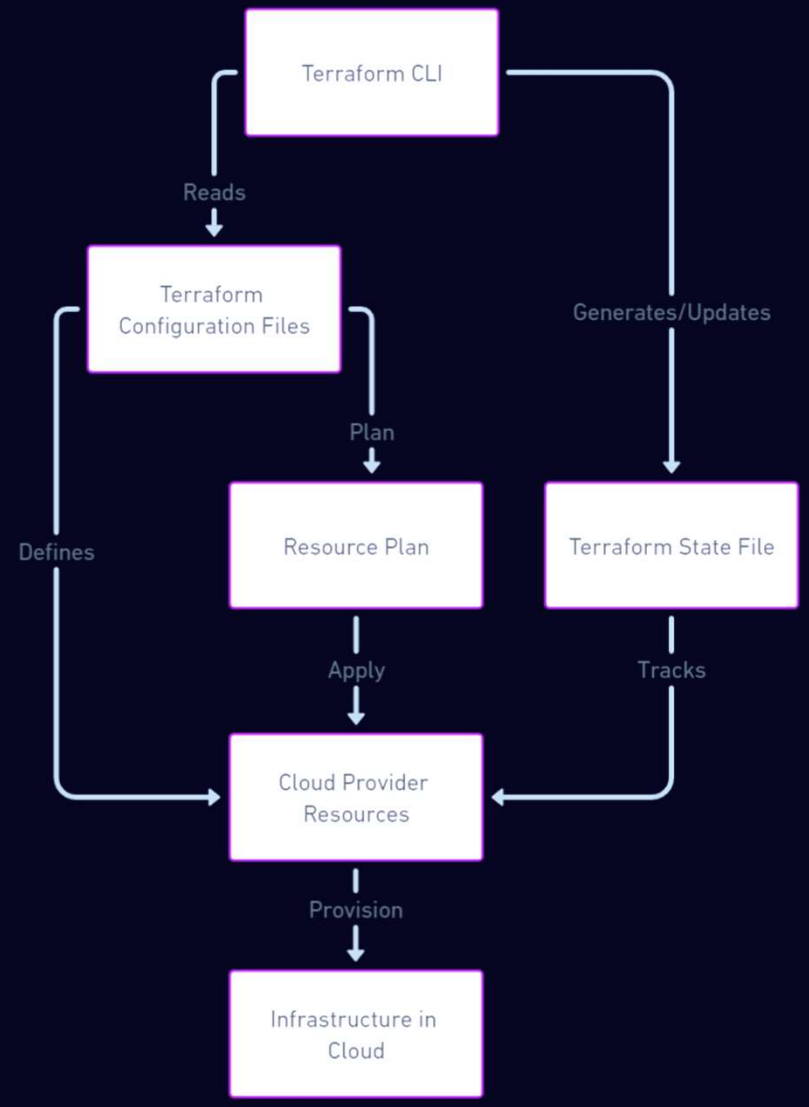
TERRAFORM

Jai

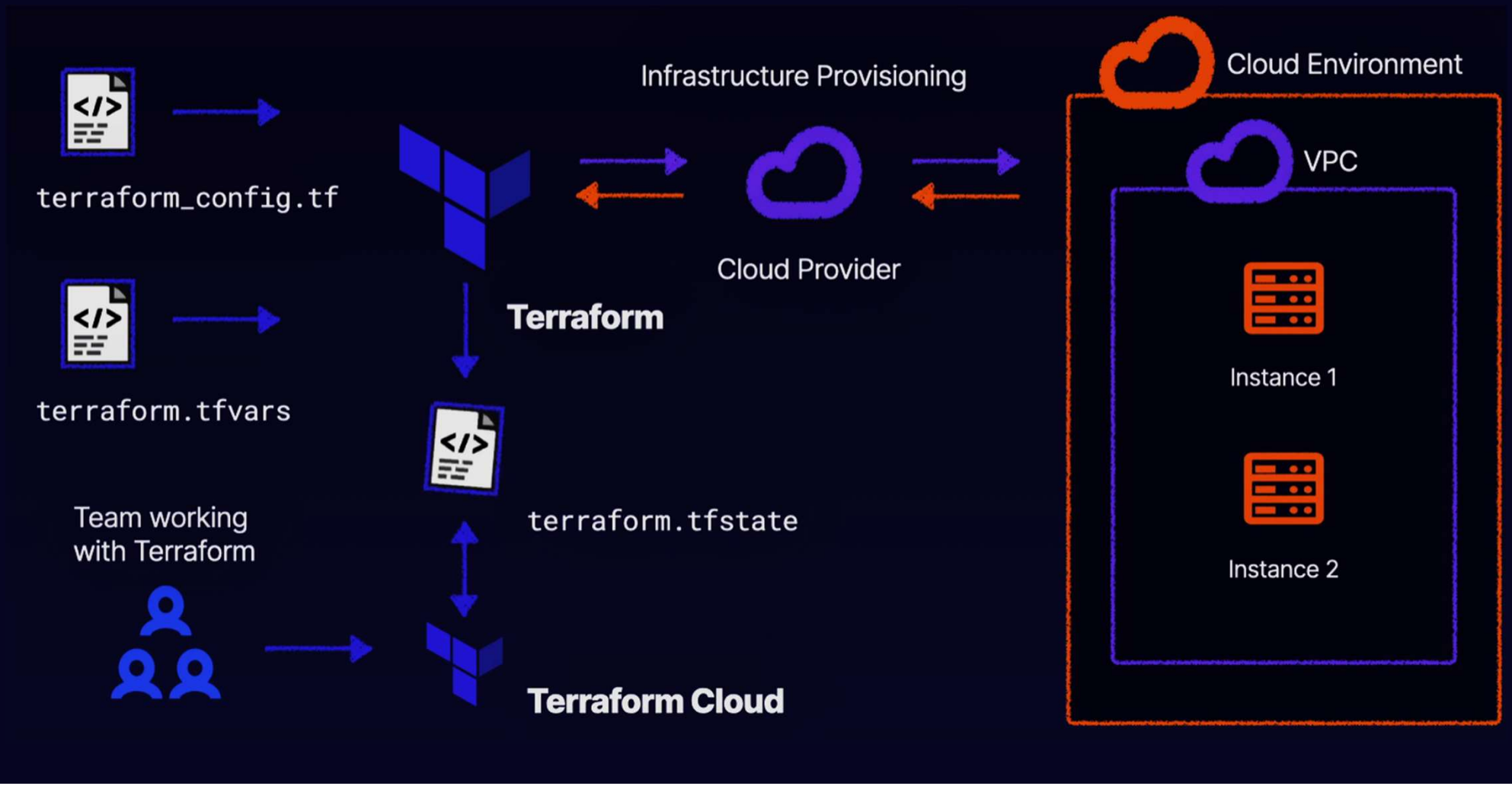
What is Terraform?

It is a tool for building, changing, and versioning infrastructure safely and efficiently, locally or in the cloud. You can manage existing and popular service providers as well as custom in-house solutions.



- The Terraform CLI reads the configuration files.
- Terraform generates a plan to show what will happen when you apply your configuration.
- The state file is updated to track the state of resources.
- The changes are applied to the cloud provider.
- The cloud provider provisions the resources, resulting in live infrastructure.

TERRAFORM Architecture



1.Infrastructure as Code: Instead of manually setting up and managing servers, databases, networks, etc., Terraform allows you to write configuration files to describe the infrastructure. Like any code, these files can be versioned and reused, which means you can track changes over time and share your setups with others.

2.Execution Plans: Before applying any changes to your infrastructure, Terraform lets you see what will happen. This "plan" is like a dry run, showing you what Terraform will do without actually doing it yet. It helps you prevent mistakes.

3.Resource Graph: Terraform visualizes how different parts of your infrastructure rely on each other. This helps to build and manage infrastructure more efficiently, ensuring that everything is created or updated in the correct order.

4.Change Automation: With Terraform, you can apply complex changes to your infrastructure automatically rather than manually. This reduces the risk of human error and ensures that changes are made in the correct order as described by the resource graph and the execution plan.

1

Infrastructure as Code

Your infrastructure is described using a high-level configuration syntax. This allows your infrastructure to be versioned and treated as you would any other code. It can also be shared and re-used.

2

Execution Plans

Terraform generates an execution plan with its “planning” step. This shows what Terraform will do when you apply the configuration. This allows you to avoid any surprises when Terraform manipulates infrastructure.

3

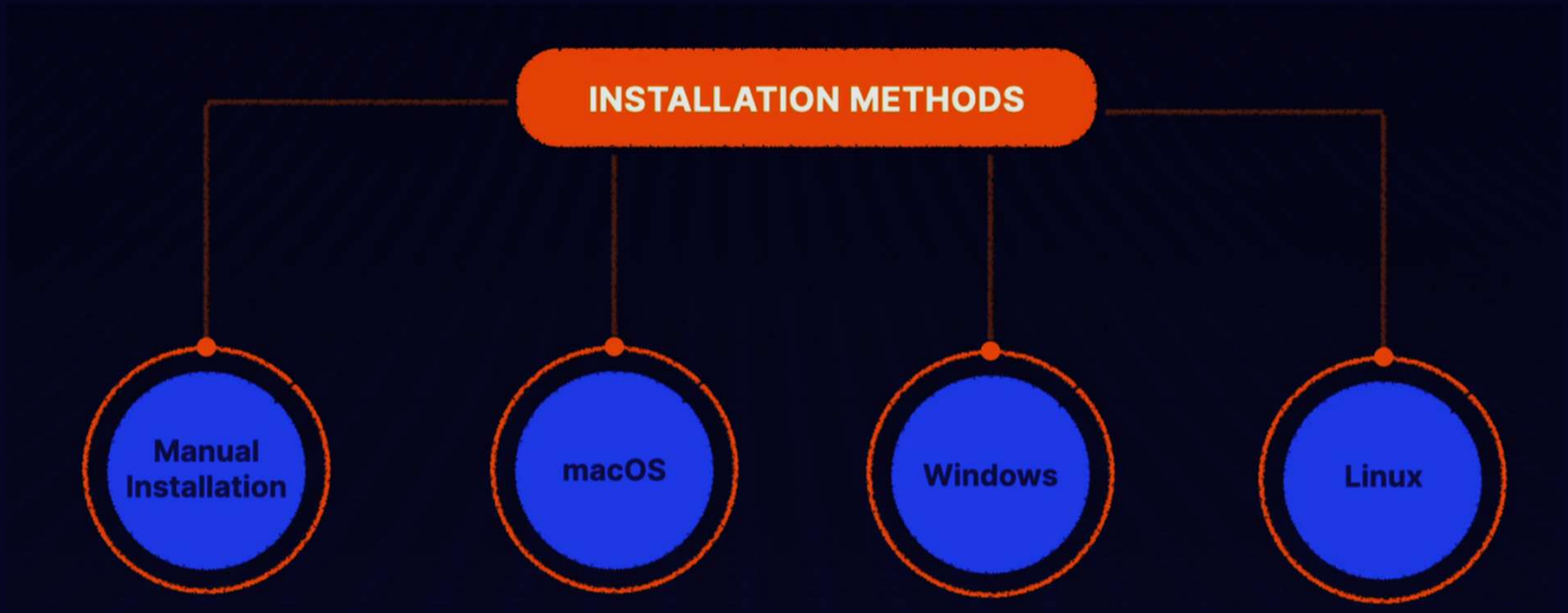
Resource Graph

Terraform builds infrastructure as efficiently as possible, and operators get insight into dependencies in their infrastructure. It accomplishes this by building a graph of all your resources, and it gives you greater insight into the creation and modification of any non-dependent resources.

4

Change Automation

Complex changes can be applied to your infrastructure with minimal interaction. With the combination of the execution plan and resource graph, you will know exactly what Terraform will change and in what order. This will help avoid many possible human errors.



https://developer.hashicorp.com/terraform/tutorials/aws-get-started/install-cli
https://developer.hashicorp.com/terraform/install

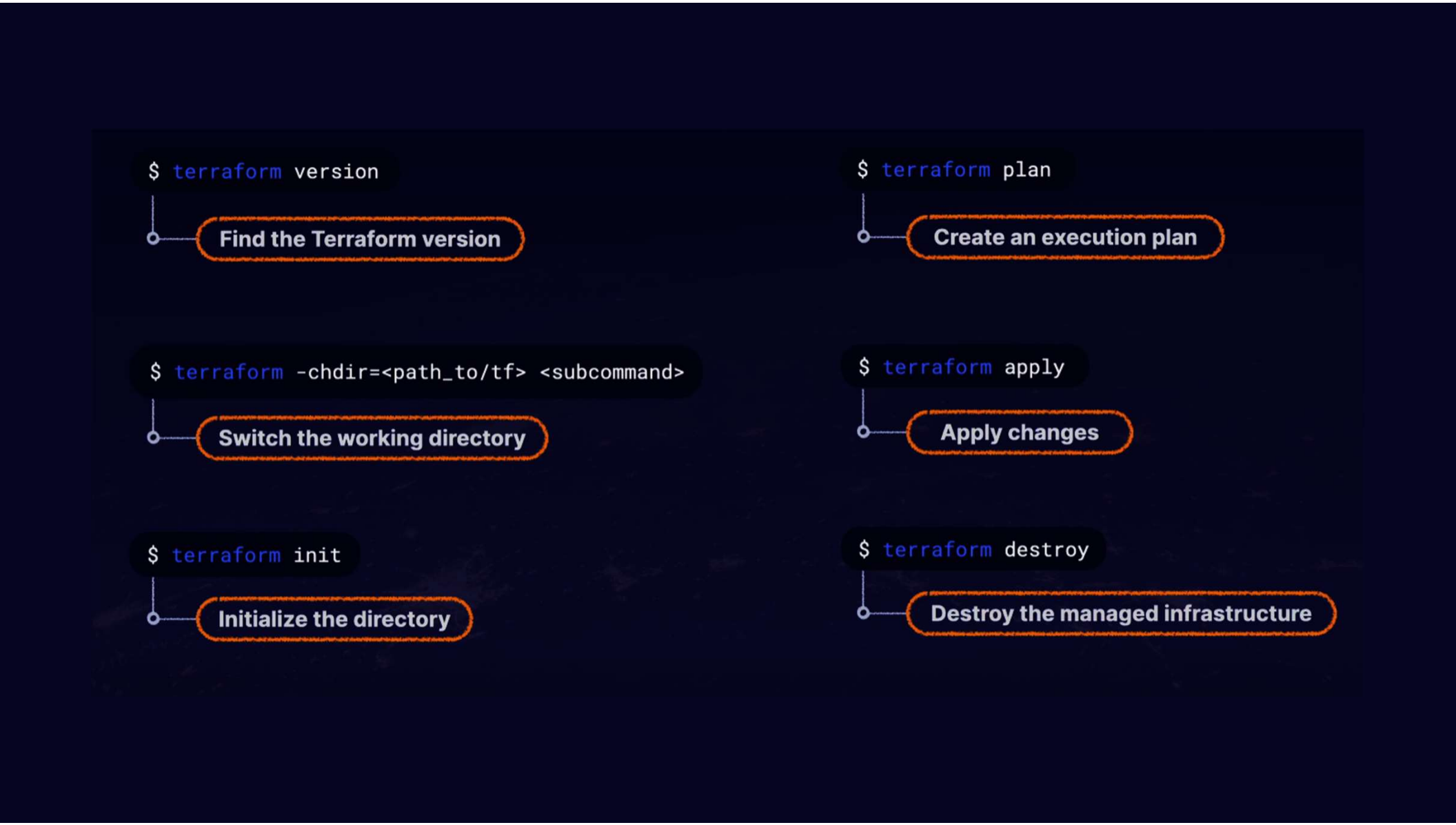
https://developer.hashicorp.com/terraform/intro

Terraform's command line interface (CLI) is accessible via the `terraform` command. This command accepts a variety of subcommands.

The `terraform` command by itself allows you to list the subcommands available.

The `terraform -install-autocomplete` command offers `bash` tab completion-like capabilities.

```
# Check the installed version of Terraform
terraform version

# Change the current directory to a specified path where your Terraform files are located before running any other Terraform command
terraform -chdir=<path_to_tf> <subcommand>

# Prepare your working directory for other commands, such as terraform apply. This may involve setting up various local settings, downloading modules, etc.
terraform init

# Show what Terraform will do without actually making any changes to real resources/services
terraform plan

# Apply the changes required to reach the desired state of the configuration
terraform apply

# Destroy all the resources that Terraform manages
terraform destroy
```


$ terraform version
Find the Terraform version
$ terraform -chdir=<path_to/tf> <subcommand>
Switch the working directory
$ terraform init
Initialize the directory
$ terraform plan
Create an execution plan
$ terraform apply
Apply changes
$ terraform destroy
Destroy the managed infrastructure

Save a Terraform deployment plan to a file
terraform plan -out <plan_name> # Replace <plan_name> with the desired name of the plan file. This will create a file that describes the changes to be made.

Create a plan for destroying all the managed infrastructure
terraform plan -destroy # This will generate a plan to remove all resources managed by Terraform.

Apply a specific deployment plan
terraform apply <plan_name> # Replace <plan_name> with the name of your saved plan file to apply those specific changes.

Apply changes to only a specific resource within the managed infrastructure
terraform apply -target=<resource_name> # Replace <resource_name> with the actual identifier of the resource you want to specifically target.

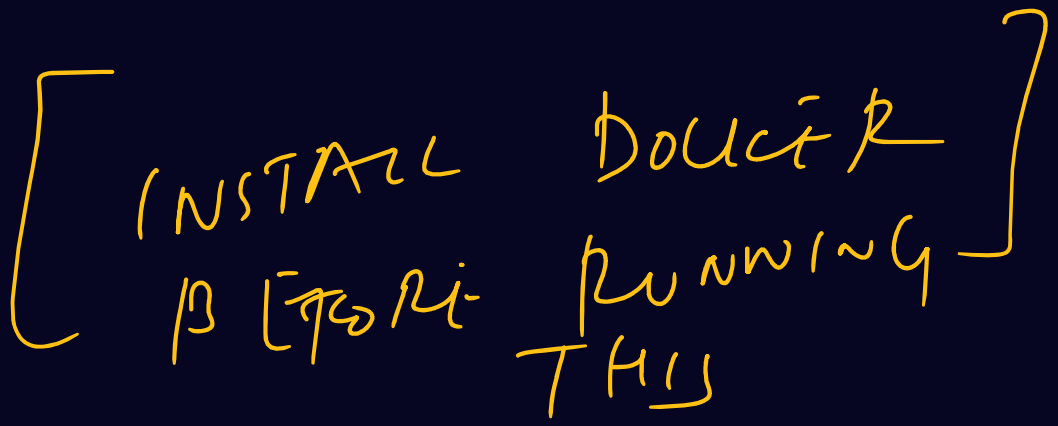
Apply changes and pass a variable value from the command line
terraform apply -var 'my_variable=<variable>' # Replace <variable> with the actual value you want to assign to 'my_variable'.

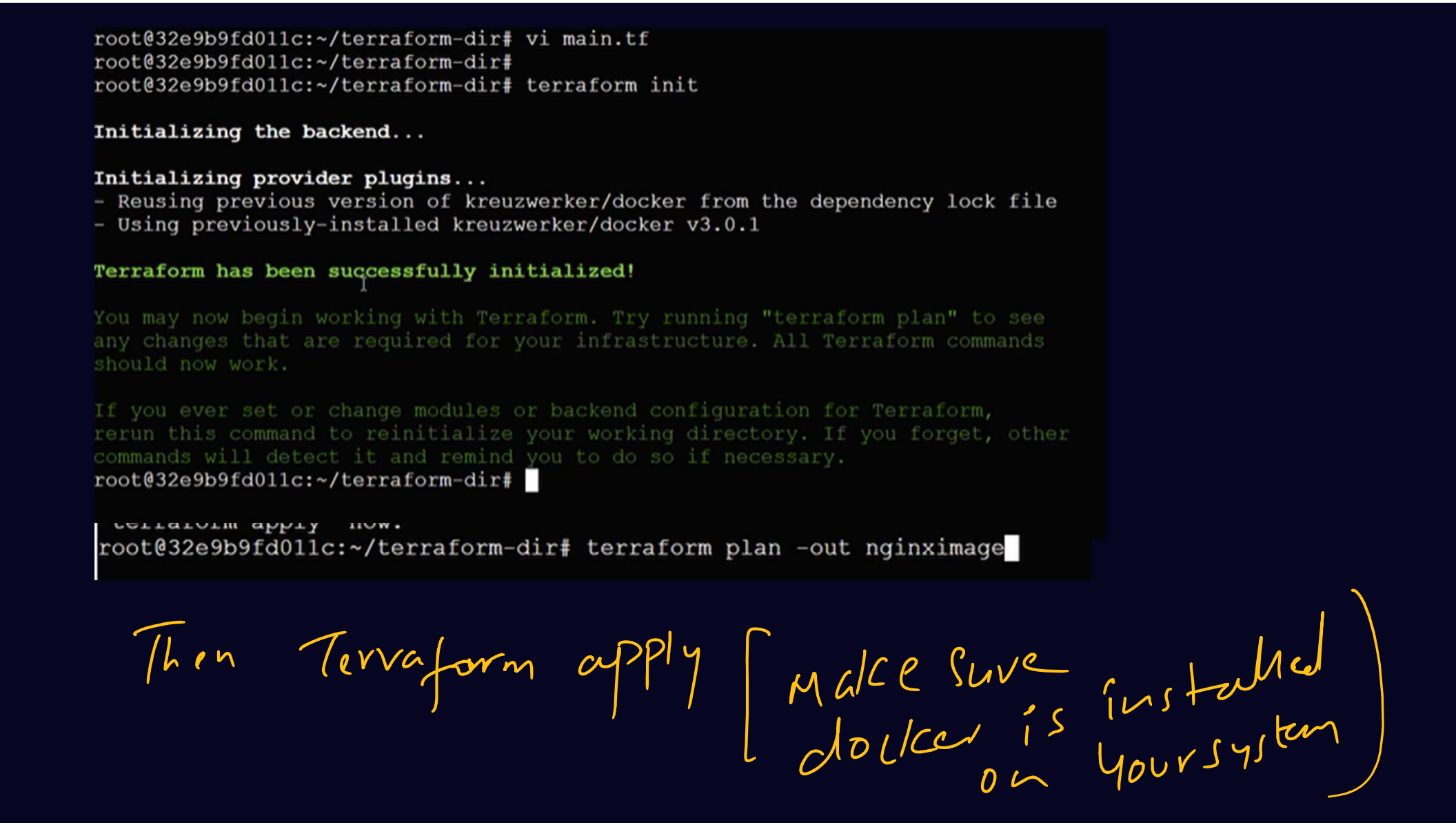
Get information about the providers used in Terraform configurations
terraform providers # This will list all the providers that Terraform is configured to use.


$ terraform plan -out <plan_name>
Output a deployment plan
$ terraform plan -destroy
Output a destroy plan
$ terraform apply <plan_name>
Apply a specific plan
$ terraform apply -target=<resource_name>
Only apply changes to a targeted resource
$ terraform apply -var my_variable=<variable>
Pass a variable via the command line
$ terraform providers
Get provider info used in configuration

```
touch main.tf

Vi main.tf

terraform {
  required_providers {
    docker = {
      source = "kreuzwerker/docker"
      version = "~> 3.0.1"
    }
  }
}

provider "docker" {}

resource "docker_image" "nginx" {
  name         = "nginx:latest"
  keep_locally = false
}

resource "docker_container" "nginx" {
  image = docker_image.nginx.image_id
  name  = "tutorial"
  ports {
    internal = 80
    external = 8000
  }
}
```

https://developer.hashicorp.com/terraform/tutorials/docker-get-started/docker-build




root@32e9b9fd011c:~/terraform-dir# vi main.tf
root@32e9b9fd011c:~/terraform-dir#
root@32e9b9fd011c:~/terraform-dir# terraform init
Initializing the backend...
Initializing provider plugins...
- Reusing previous version of kreuzwerker/docker from the dependency lock file
- Using previously-installed kreuzwerker/docker v3.0.1
Terraform has been successfully initialized!
You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.
If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
root@32e9b9fd011c:~/terraform-dir#
root@32e9b9fd011c:~/terraform-dir# terraform plan -out nginximage
Then Terraform apply [make sure docker is installed on yoursystem]

```
root@32e9b9fd011c:~/terraform-dir# docker ps -a
CONTAINER ID   IMAGE          COMMAND                  CREATED          STATUS                      PORTS                    NAMES
dface25f2ad4   3f8a00f137a0   "/docker-entrypoint.…"   9 seconds ago    Up 7 seconds                0.0.0.0:8000->80/tcp     tutori
93dbf190dacf   hello-world    "/hello"                 27 minutes ago   Exited (0) 27 minutes ago                            modest
manujan
root@32e9b9fd011c:~/terraform-dir# curl http://localhost:8000
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
html { color-scheme: light dark; }
body { width: 35em; margin: 0 auto;
font-family: Tahoma, Verdana, Arial, sans-serif; }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
```

Destroy using → terraform destroy [IT creates tfstate file and backup will cover this in coming sessions]

Configuration Language

Blocks: These are the basic structures that describe the various configurations in Terraform. Think of them like containers that hold related settings for a specific purpose.
Example: A resource block is used to manage a particular type of resource (like a server, database, etc.) within a provider (like AWS, Google Cloud, etc.).

```
# A block defining an AWS VPC resource with a specific CIDR block
resource "aws_vpc" "main" {
  cidr_block = var.base_cidr_block
}
```

Arguments: These are the settings or properties within a block. They assign a specific value to a name, configuring aspects of the resource or provider.
Example: In a VPC resource block, cidr_block is an argument that specifies the IP address range for the VPC.

```
cidr_block = var.base_cidr_block # Argument that sets the CIDR block for the VPC
```

Expressions: These are used to assign values to the arguments. They can be simple literals, like a string or number, or they can be more complex, involving functions and references to other resources or variables.
Example: var.base_cidr_block is an expression that might refer to a variable defined elsewhere in the Terraform configuration that holds the value of the VPC's CIDR block.

```
var.base_cidr_block # Expression that represents the value of the CIDR block
```

https://developer.hashicorp.com/terraform/language

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

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

erraform language's main purpose is to declare resources. This represents infrastructure objects. All the different features are present to accommodate more flexible and convenient resource definition.

Syntax Consists Of:

- *Blocks* are containers for objects like resources.
- *Arguments* assign a value to a name.
- *Expressions* represent a value.

1.Purpose of the Terraform Language:

1. The primary objective of Terraform language is to declare resources, which are representations of infrastructure objects within the configuration.
2. Other language features serve to enhance flexibility and convenience in defining resources.

2.Structure of Terraform Configuration:

1. Terraform configuration is a comprehensive document in the Terraform language, detailing how to manage a specific collection of infrastructure.
2. It can consist of multiple files and directories.

3.Basic Elements of Terraform Configuration Syntax:

1. Blocks: Containers for objects like resources, which are organized hierarchically.
2. Arguments: Assign values to names within blocks.
3. Expressions: Represent values, either literally or by referencing other values.

4.Declarative Nature of Terraform Language:

1. Terraform configuration is declarative, describing the desired state rather than the sequence of steps to achieve it.
2. The ordering of blocks and files is typically insignificant, with Terraform determining the operation order based on implicit and explicit resource relationships.

Directories and Files

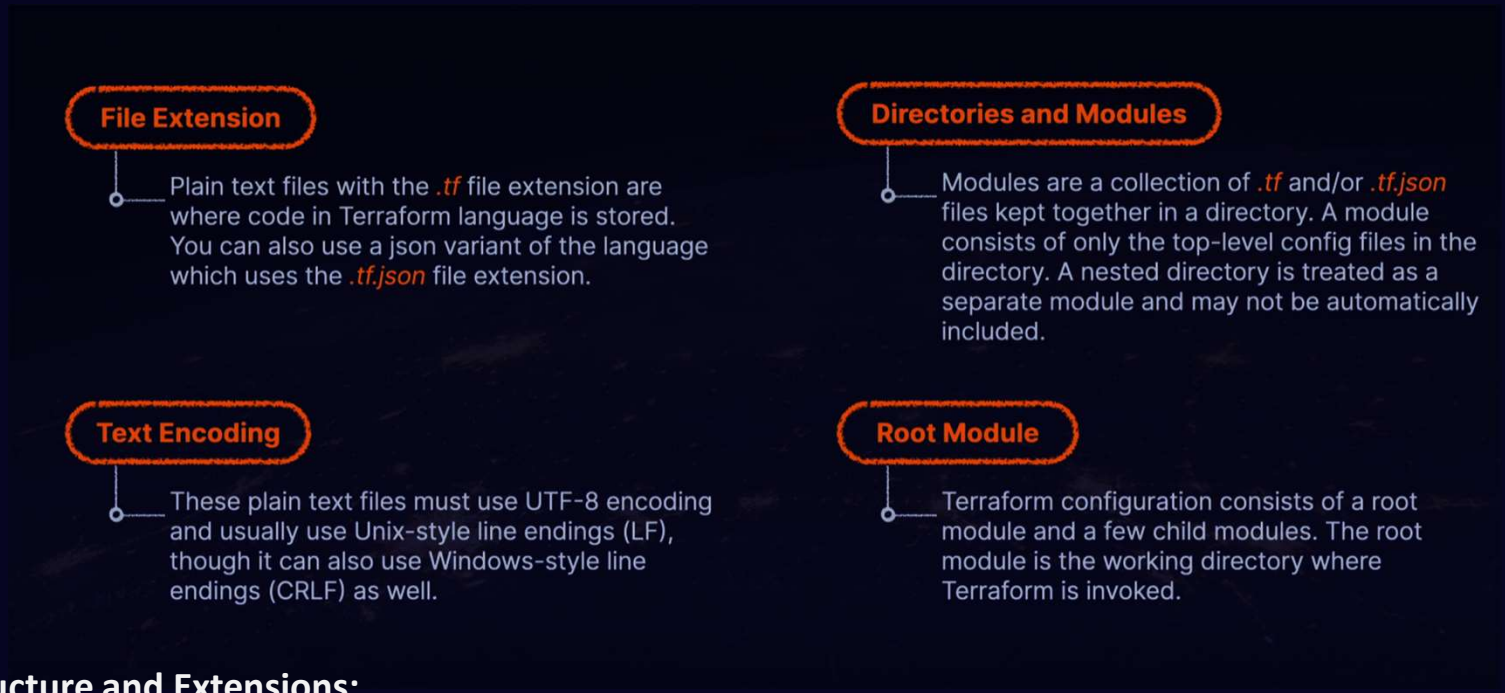
File Extension: Terraform uses files ending in .tf to store its configuration code. This code is written in the Terraform language, which describes your infrastructure setup. There's also a JSON format available, which uses .tf.json for its files, allowing for configurations to be written in JSON.
Example: A file named main.tf could contain the setup for a cloud server.

Text Encoding: The .tf or .tf.json files are plain text and should be saved using UTF-8 encoding. This ensures that the characters in the files are understood correctly across different systems and platforms. They typically use Unix-style line endings (LF), but can also handle Windows-style line endings (CRLF).
Example: If you open a file named variables.tf in a text editor, it should display the text correctly without any strange characters, assuming it's encoded in UTF-8.

Directories and Modules: In Terraform, a module is a container for multiple .tf (or .tf.json) files that are placed in the same directory. These files together define a related set of resources and settings. Nested directories are treated as separate modules, which means they are not automatically part of the parent module.
Example: You could have a directory named network with files like vpc.tf and subnets.tf, which together define your cloud network setup. This directory would be considered a module.

Root Module: When you run Terraform commands, it works within a root module. This is simply the main directory where you run Terraform, and it often contains references to child modules, which are separate directories with their own .tf files.
Example: If you have a Terraform project with a main.tf at the top level and other directories/modules like network or database, the directory containing main.tf is the root module. When you run terraform apply, it starts working in this root module.

Directories and Files



1.File Structure and Extensions:

1. Terraform files have specific extensions: .tf for main language files and .tf.json for JSON variant.
2. Text encoding must be UTF-8, and line endings can be Unix or Windows style.

2.Directories and Modules:

1. Terraform configuration comprises directories and modules.
2. Modules are collections of .tf and/or .tf.json files, treated as top-level config files.
3. Nested directories are treated as separate modules and may require explicit inclusion in the configuration.

Override Files

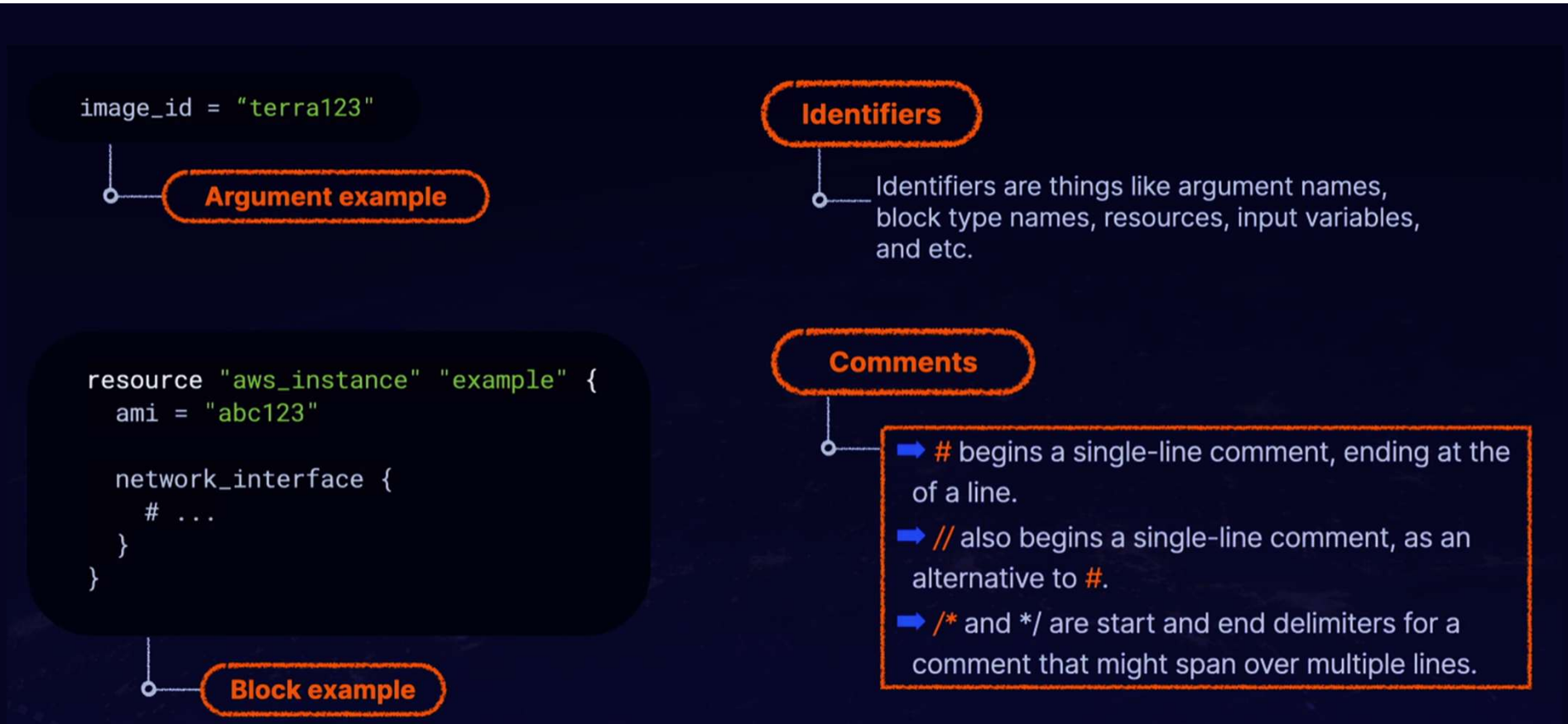
Override Files

Resource Overrides:

- Overrides can be used in special cases to modify resource configurations.
- Overrides are handled specially by Terraform, with one example provided for clarity.

Comments

Syntax and Constructs:



Syntax and Constructs:

•Terraform language syntax, called HCL (HashiCorp Configuration Language), is designed for readability and ease of writing.

•Two key syntax constructs: arguments and blocks, with examples provided for clarity.

•Identifiers follow specific rules and support different types of comments.

JSON Syntax Option:

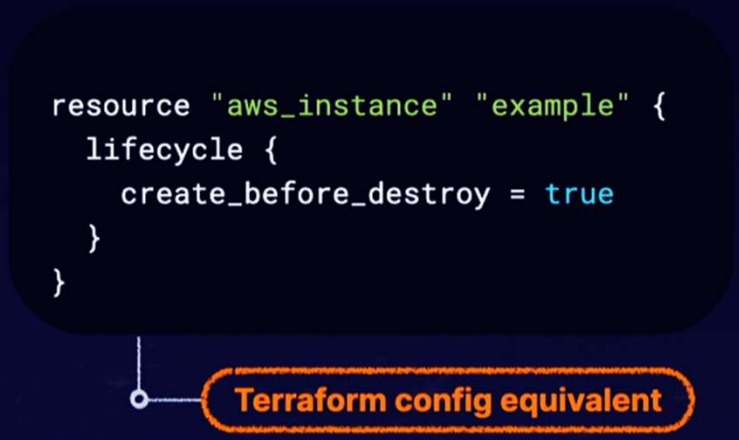
1.JSON Syntax Option:

1. Terraform configurations can also be expressed in JSON syntax, but it may add complexity compared to native syntax.
2. Examples are provided comparing JSON and native syntax.

2.Choosing Syntax:

1. While both JSON and native syntax are supported, simplicity often favors native Terraform language.
2. Users can choose the syntax they're comfortable with or use a combination of both.

JSON Config vs Terraform Config



Working with Resources

https://developer.hashicorp.com/terraform/language/resources
https://developer.hashicorp.com/terraform/language/meta-arguments/depends_on

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

Resources are the most important part of the Terraform language. Resource blocks describe infrastructure objects like virtual networks, compute instances, or components like DNS records.

Resource Types:

- ***Providers***, which are plugins for Terraform that offers a collection of resources types.
- ***Arguments***, which are specific to the selected resource type.
- ***Documentation***, which every provider uses to describe its resource types and arguments.

1.Importance of Resources:

1. Resources are vital in Terraform configurations, defining infrastructure objects like virtual networks, compute instances, and DNS records.

2.Resource Blocks:

1. Declare infrastructure resources with a specific type and local name.
2. The name serves as an identifier within the module and must be unique.
3. Configuration arguments within the block are specific to the resource type.

3.Providers:

1. Plugins offering resource types for managing cloud or on-premise infrastructure.
2. Terraform automatically installs most providers.
3. Modules must specify required providers and may require configuration for remote API access.

4.Arguments:

1. Specific to selected resource type, documented in the provider's documentation.
2. Values can utilize expressions and dynamic Terraform features.

Meta-Arguments

Meta-Arguments

depends_on

Specify hidden dependencies.

provider

Select a non-default provider configuration.

count

Create multiple resource instances according to a count.

lifecycle

Set lifecycle customizations.

for_each

Create multiple instances according to a map or a set of strings.

provisioner and connection

Take extra actions after resource creation.

depends_on: This is used to define explicit dependencies between resources. Sometimes Terraform can't automatically detect that one resource needs to be created before another, and you use depends_on to tell it explicitly.
Example: If you have a database that should only be created after a server is up and running, you can use depends_on in the database resource to explicitly depend on the server resource.

count: This keyword lets you create multiple instances of a resource without having to copy and paste the configuration. You specify a number and Terraform creates that many copies of the resource.
Example: If you want to create 10 identical servers, you can use count and set it to 10 in your server resource configuration.

for_each: Similar to count, but instead of a number, you use a map or a set of strings, and Terraform creates an instance for each item in the set or map.
Example: If you have a set of different server names, you can use for_each to create a server for each name in your set.

provider: This specifies which provider (like AWS, Google Cloud, etc.) Terraform will use to create resources. You can select a non-default configuration if you have multiple configurations for the same provider.
Example: If you have two AWS accounts, you can use provider to specify which account's configuration to use when creating a resource.

lifecycle: This block within a resource allows you to customize how Terraform manages the resource throughout its lifecycle, such as preventing accidental deletion or changing how updates are handled.
Example: If you want a server to be replaced completely every time it changes rather than updated in place, you could set this behavior in a lifecycle block.

provisioner and connection: Provisioners are used to execute actions on the local machine or on a remote machine in order to set up a resource. connection settings are used within provisioners to specify how to connect to the remote resource, typically with SSH or WinRM.
Example: After a server is created, you might use a provisioner to copy files onto it or execute a script, and connection to define how Terraform should connect to the server to run these commands.

Timeouts

In Terraform, when you create, update, or delete resources, these operations can sometimes take a long time. The timeouts feature allows you to set maximum durations for these operations. If an operation takes longer than the specified timeout, Terraform will consider it as failed and stop the process. This can help prevent a Terraform run from hanging indefinitely due to some resources taking too long to provision or delete.

In the example provided in the image:

create = "60m": This sets a maximum time of 60 minutes for creating the resource. If the resource is not created within 60 minutes, Terraform will stop trying and mark it as an error.

delete = "2h": This sets a maximum time of 2 hours for deleting the resource. If the resource is not deleted within 2 hours, Terraform will stop trying and mark it as an error.

The main purpose is to have control over how long Terraform should wait for an operation to complete before giving up. This is useful in automated environments where you want to avoid indefinite delays or in situations where you expect an operation to complete within a certain time frame.

Timeouts:
- Allow customization of operation duration.
- Specified in create, update, and delete operations.
- Values include minutes, seconds, and hours.

```
resource "aws_db_instance" "example" {
  # ...

  timeouts {
    create = "60m"
    delete = "2h"
  }
}
```

There are some resource types that provide special *timeouts*, nested block arguments that allow for customization of how long certain operations are allowed to take before they are deemed failed.

***timeouts* string examples**

- "60m"
- "10s"
- "2h"

1.Accessing Resource Attributes:

1. This refers to the use of expressions in Terraform to get information about the attributes of a resource. For example, after you create a server, you might want to get its IP address.
2. Read-only attributes are pieces of information that Terraform gets from the cloud provider after a resource is created, such as an ID or an endpoint that the cloud service generates.
3. Provider-included data sources are special Terraform configurations that fetch data from a provider's API without managing any resources. They are used to read information that can be used in other parts of your Terraform code.

2.Resource Dependencies:

1. Terraform automatically figures out what order to create resources in based on their dependencies. For example, you can't create a database record before the database itself exists.
2. Expressions in one resource that reference attributes of another resource help Terraform understand the order in which resources should be created, updated, or deleted.
3. While Terraform is good at figuring out these dependencies on its own, sometimes you may need to specify them manually if Terraform can't detect them.

3.Local-only Resources:

1. These are special resources that don't manage any real-world entity but are used within Terraform to perform local operations. An example is generating random IDs or SSH keys that are not created on any cloud provider but are needed by other resources within Terraform

Accessing Resource Attributes:

```
resource "aws_instance" "web_server" {
  # ...
}

output "web_server_ip" {
  value = aws_instance.web_server.public_ip
}
```

Here, aws_instance.web_server.public_ip is a read-only attribute you're accessing.

Resource Dependencies:

```
resource "aws_db_instance" "my_database" {
  # Terraform will ensure this is created before any resources that depend on it.
  # ...
}

resource "aws_instance" "web_server" {
  # This implicitly depends on `my_database` because it uses its address.
  user_data = <<-EOF
              #!/bin/bash
              echo "Database Address: ${aws_db_instance.my_database.address}"
              EOF
  # ...
}
```

Here, aws_instance implicitly depends on aws_db_instance because it uses its address in the user_data script.

Local-only Resources:

```
resource "random_id" "server_id" {
  # This will generate a random ID used by the server resource as part of its name.
  byte_length = 8
}

resource "aws_instance" "web_server" {
  tags = {
    Name = "web-server-${random_id.server_id.hex}"
  }
  # ...
}
```

Here, random_id.server_id.hex generates a local random string that you can use to uniquely tag your server in AWS.

Accessing Resource Attributes

- Expressions within Terraform modules
- Read-only attributes with information obtained from remote APIs
- Provider-included data sources

Local-only Resources

Specialized resource types that operate only within Terraform itself, calculating some results and saving those results in the state for future use.

Examples:

- ssh keys
- self-signed certs
- random ids.

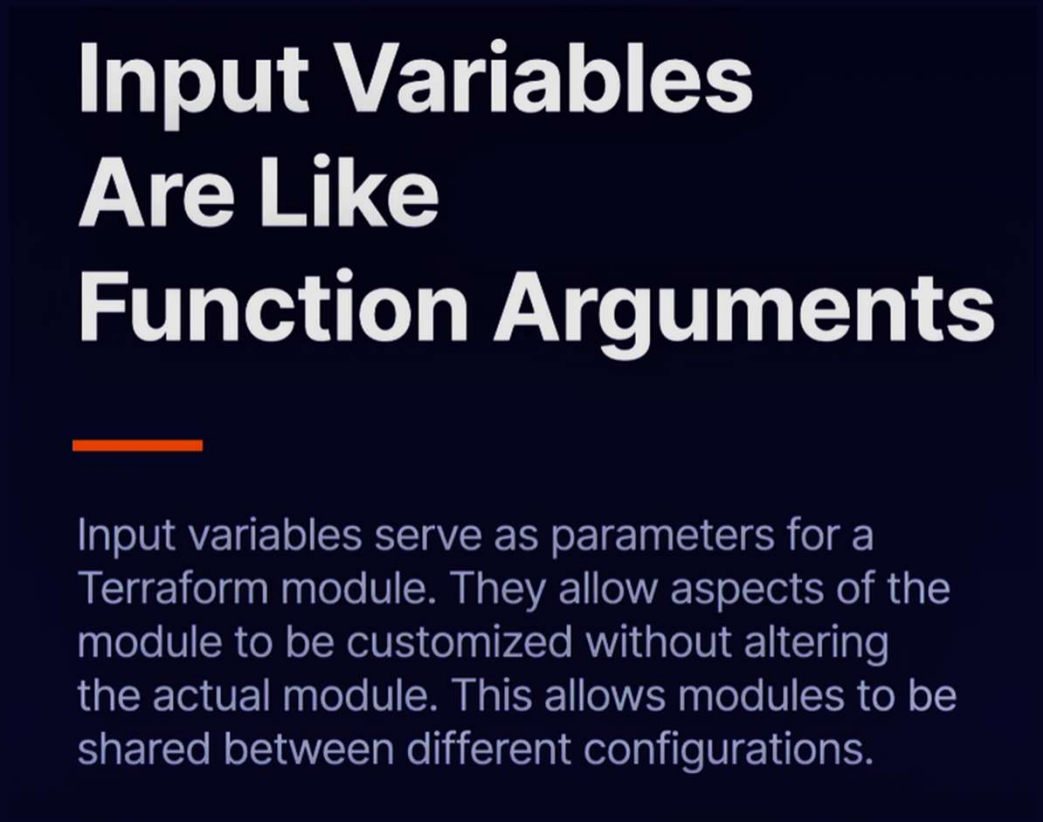
Resource Dependencies

- Most resource dependencies are handled automatically.
- Terraform analyzes any expressions within a resource block to find references to other objects, and treats those references as ordering requirements when creating, updating, or destroying resources.
- It's usually not necessary to manually specify dependencies between resources, however some dependencies cannot be recognized implicitly in the configuration.

Input Variables

Input Variables

https://developer.hashicorp.com/terraform/language/values/variables



input variables as being akin to function arguments. In programming, function arguments allow for the passing of different values to a function to get different results without changing the function's code. Similarly, input variables in Terraform allow modules to be reused with different configurations without altering the module's core code. For instance, by changing the image_id variable, you can reuse the same Terraform code to provision different types of virtual machines in a cloud environment.

```
variable "server_image" {
  description = "The image ID to use for the server."
  type        = string
}

variable "server_type" {
  description = "The type of server to create."
  type        = string
  default     = "t2.micro"
}

variable "region" {
  description = "The region to launch the server."
  type        = string
  default     = "us-west-1"
}
```

Declaring an Input Variable

```
variable "image_id" {
  type = string
}

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

variable "docker_ports" {
  type = list(object({
    internal = number
    external = number
    protocol = string
  }))
  default = [
    {
      internal = 8300
      external = 8300
      protocol = "tcp"
    }
  ]
}
```

Variable Block Example

The name of a variable can be any valid ***identifier*** except for *source*, *version*, *providers*, *count*, *for_each*, *lifecycle*, *depends_on*, and *locals*.

Variable Definition Syntax in Terraform:

Variables in Terraform are declared using the variable keyword and are used to parameterize Terraform configurations. For example:

image_id is a string variable with no default value, which means it must be provided by the user.

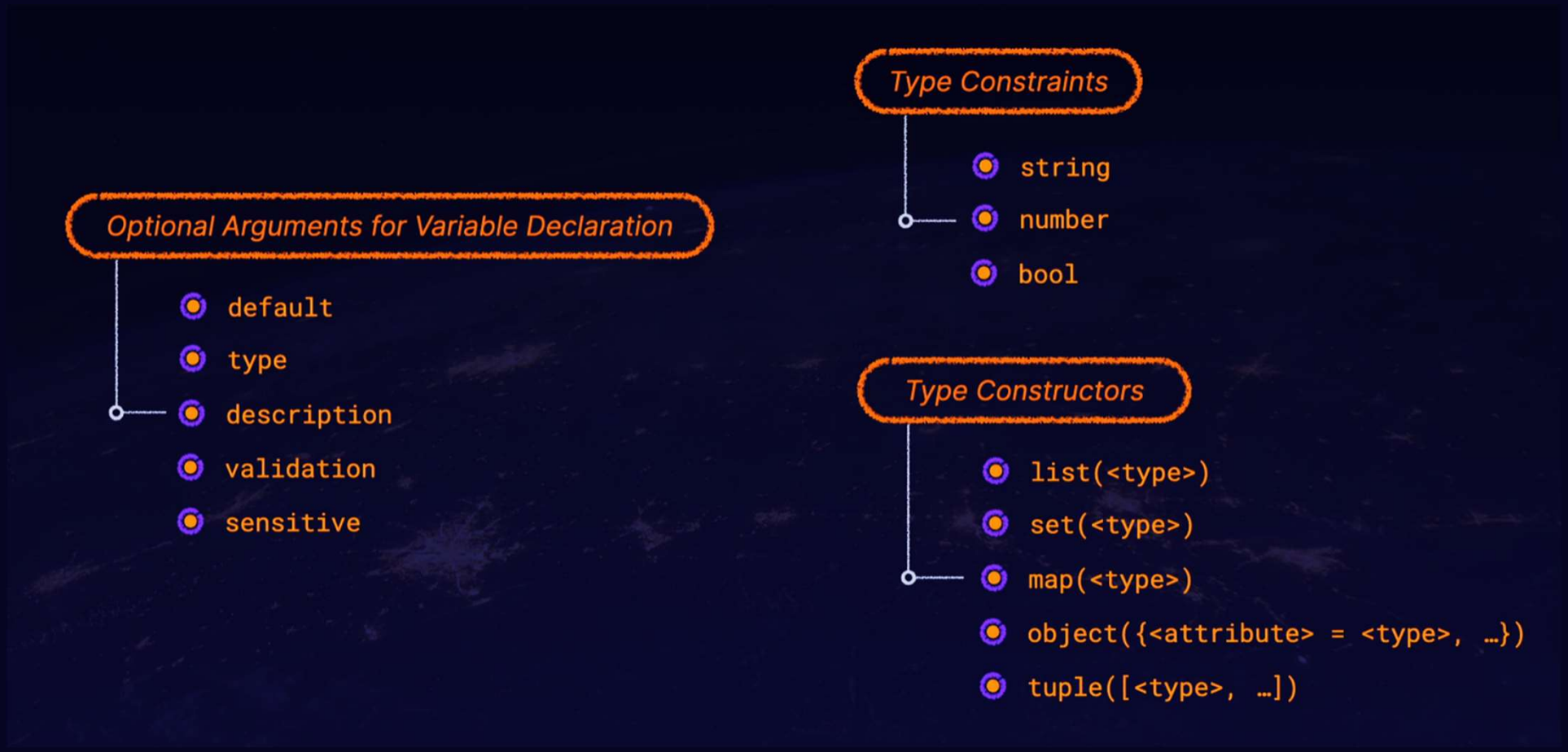
availability_zone_names is a list of strings with a default value of ["us-west-1a"]. This means if the user does not specify this variable, Terraform will use us-west-1a as the default availability zone.

docker_ports is a more complex variable where its type is a list of objects. Each object has an internal and external number representing port numbers, and a protocol string. A default value for this variable is provided, setting both internal and external ports to 8300 with the TCP protocol.

Naming Variables:

Variable names can be any valid identifier except for reserved keywords such as source, version, providers, count, for_each, lifecycle, depends_on, and locals. These reserved keywords have special meanings in Terraform and cannot be used as variable names.

Arguments and Constraints



Optional Arguments for Variable Declaration:

1. default: This provides a default value for the variable if no value is input by the user. For instance, you might have a variable for a region with a default value of "us-east-1".

2. type: Specifies the data type of the variable. It could be a string, number, bool, etc. For example, a variable for instance size might be a string, like "large".

3. description: A string that explains the purpose of the variable, which can be very helpful for others using your Terraform module. A description for a database password variable might be "The password for the database server.“

4. validation: Allows you to enforce certain conditions that the input value of a variable must meet. For example, you can validate that a number is within a certain range.

5. sensitive: When set to true, it instructs Terraform to treat the variable's value as sensitive information and to hide it from the console output when Terraform runs. This would be used for passwords or API keys.

Type Constraints:

These specify the kind of data that a variable is expected to contain. The simple type constraints are string, number, and bool. An example of a type constraint is specifying that a variable that holds an instance type must be a string.

Type Constructors:

1. These are more complex types that allow for structuring data.

2. list(<type>): A sequence of values of a single type. For example, a list of subnet IDs could be defined as list(string).

3. set(<type>): Similar to a list but enforces uniqueness of the items. For instance, a set of IP addresses can be a set(string).

4. map(<type>): A collection of values where each is identified by a string label, similar to a dictionary or a hash table in other languages. You could have a map for storing environment variables where the keys are the variable names and the values are the variable contents.

5. object({<attribute> = <type>, ...}): Defines a structure with a fixed set of named attributes and their types. An example could be defining an object for a VM with attributes like name, image, and size.

6. tuple([<type>, ...]): A sequence of elements, potentially of different types, fixed in size and order. An example could be a tuple representing a 2D coordinate with a number for the x value and another for the y value.

Argument Examples

Input Variable Description

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

Custom Validation Rules

```
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-\"."
  }
}
```

Using the Sensitive Argument

```
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
}
```

Resulting Terraform Output

```
Terraform will perform the following
actions:

  # some_resource.a will be created
  + resource "some_resource" "a" {
      + name    = (sensitive)
      + address = (sensitive)
    }

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

Using Input Variable Values

A value can be accessed from an expression, such as `var.<var_name>`.

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

A value assigned to a variable can only be accessed in an expression within the module it was declared.

How to Assign Values to Root Modules Variables

How to assign values:

- In a Terraform Cloud workspace
- Individually, with the `-var` command line option
- In variable definitions files like `.tfvrs` or `.tfvars.json`
- As environment variables

Variables on the command line:

```
$ 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"}'
```

How to Assign Values to Root Modules Variables

Calling the variable definition file from the command line:

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

Automatically loaded definition files:

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

Variable definition file example:

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

Environment Variables

Terraform searches the environment for environment variables named `TF_VAR_` followed by the name of the variable.

```
$ export TF_VAR_image_id=ami-abc123
$ terraform plan
...
```

Terraform matches the variable name exactly as given in configuration on operating systems where the environment variable is case-sensitive. The required environment variable name usually includes a mix of upper and lowercase letters, like in the example above.

Precedence

The order in which variables are loaded:

- Environment variables
- terraform.tfvars
- terraform.tfvars.json
- *.auto.tfvars or *.auto.tfvars.json
- Any command-line options like -var and -var-file

Declaring Output Variables

https://developer.hashicorp.com/terraform/language/values/outputs

Output Variables Are Like Return Values

They have many uses:

- ✓ A child module can use them to expose a subset of resource attributes to the parent module.
- ✓ A root module can use them to print values in the CLI.
- ✓ Root module outputs can be accessed by other configurations via the `terraform_remote_state` data source.

Declaring an Output Variable

Output Block Example

```
output "instance_ip_addr" {
  value = aws_instance.server.private_ip
}
```

The `name` label after the `output` keyword must be a valid identifier.

The `value` argument takes an expression whose result will be returned to the user.

Arguments and Constraints

When accessing child module outputs in a parent module, the outputs of the child modules are available in expressions as `module.<module_name>.<output_name>`.

Argument Examples

Using the `description` Output Argument

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

The `description` should clearly explain the purpose of the `output` and what kind of `value` is expected.

Argument Examples

Using the sensitive Argument

```
# main.tf

module "foo" {
  source = "./mod"
}

resource "test_instance" "x" {
  some_attribute = module.mod.a
}

output "out" {
  value     = "xyz"
  sensitive = true
}

output "a" {
  value     = "secret"
  sensitive = true
}
```

Resulting Terraform Output

```
Terraform will perform the following
actions:

  # test_instance.x will be created
  + resource "test_instance" "x" {
      + some_attribute    = (sensitive)
    }

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

Changes to Outputs:
  + out = (sensitive value)
```

Using the depends_on Output Argument

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

  depends_on = [
    # Security group rule must be created before this IP address could
    # actually be used, otherwise the services will be unreachable.
    aws_security_group_rule.local_access,
  ]
}
```

The depends_on argument should be used as a last resort. You should always include a comment explaining why it is being used.

Declaring Local Variables

Local Values Are Like a Function's Temporary Local Variables

Allow you to:

- ✓ Assigns a name to an expression.
- ✓ Use the variable multiple times within a module without repeating it.

Declaring a Local Variable

Locals Block Example

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

A set of related **local** values can be declared together in a single block.

The **expressions** are not limited to literal constants; they can reference other values in the module.

Using Local Values

When a local value is declared, you can reference it in expressions as `local.<name>`.

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

  tags = local.common_tags
}
```

Local values can only be accessed in expressions within the module where they were declared.

Data Sources

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

```
data "aws_vpc" "selected" {
  default = true
}
```

In this example, the `aws_vpc` data source is used to fetch the default VPC in the AWS account. The `default` attribute is set to `true` to specify that we want the default VPC.

Once this data source is defined, you can reference its values elsewhere in your Terraform configuration. For example, you could use it to create a subnet in this VPC:

```
resource "aws_subnet" "example" {
  vpc_id     = data.aws_vpc.selected.id
  cidr_block = "10.0.1.0/24"
}
```

In this example, `data.aws_vpc.selected.id` is used to set the `vpc_id` attribute of the `aws_subnet` resource. This means that the subnet will be created in the VPC that was fetched by the `aws_vpc` data source.

Import Block

In Terraform, the `import` command is used to bring resources that were created outside of Terraform under Terraform's management. This is useful when you have existing infrastructure that you want to manage with Terraform.

The general syntax for the `import` command is:

```
terraform import <resource_type>.<resource_name> <resource_id>
```

Here's a simple real-world example. Suppose you have an AWS S3 bucket that was created outside of Terraform, and you want to bring it under Terraform's management.

First, you would define a resource block for the S3 bucket in your Terraform configuration:

```
resource "aws_s3_bucket" "bucket" {
  # Note: bucket name is not specified here
}
```

Then, you would use the `terraform import` command to import the existing S3 bucket. The `<resource_id>` is the name of the S3 bucket:

```
terraform import aws_s3_bucket.bucket my-existing-bucket
```

After running this command, the S3 bucket `my-existing-bucket` is now managed by Terraform. The state of the bucket is imported into the Terraform state file, and you can manage the bucket using Terraform.

Please note that `terraform import` does not generate configuration. After importing the resource, you must manually write the configuration for the imported resource to match its current state.

Modules

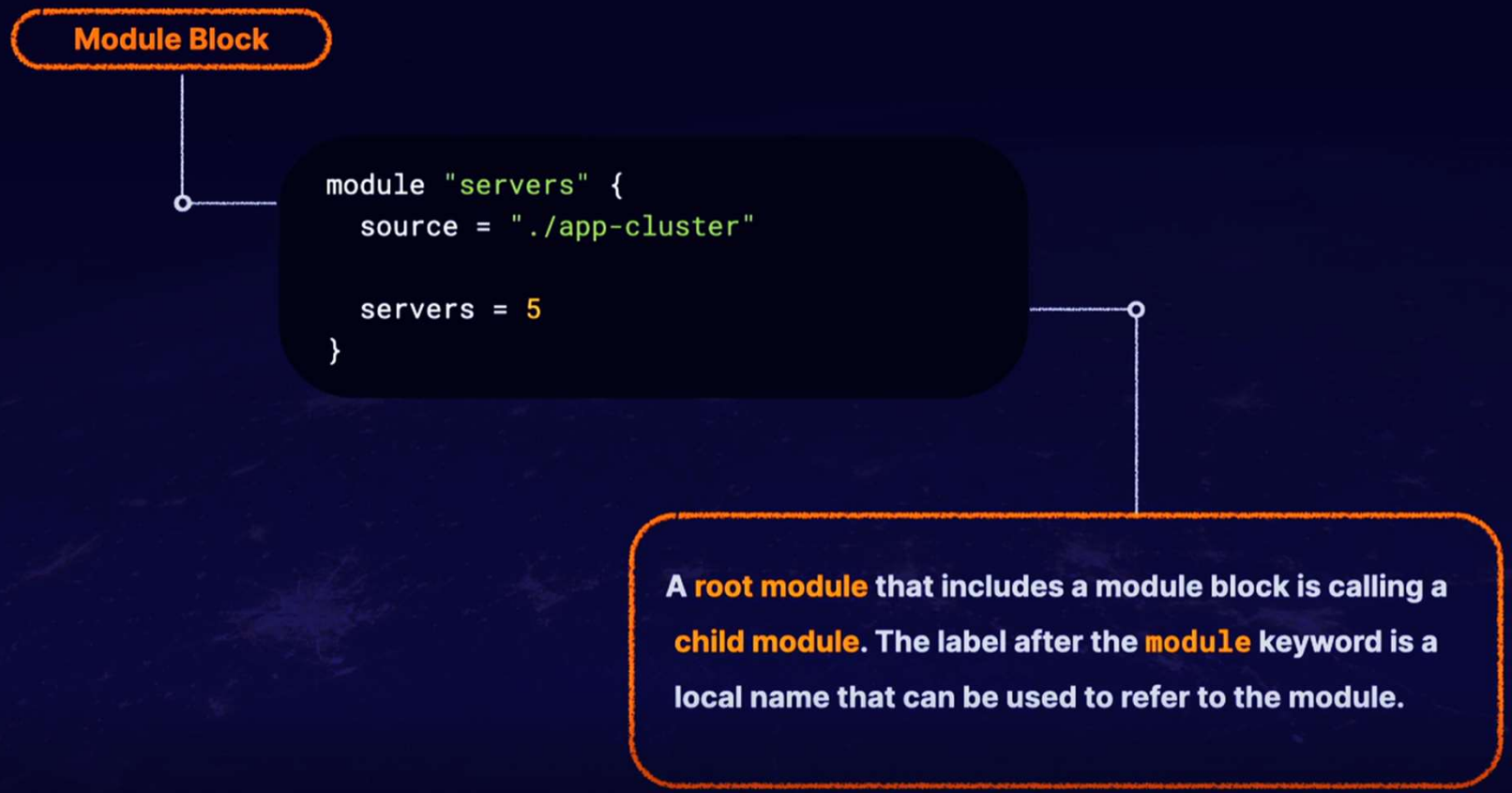
https://developer.hashicorp.com/terraform/language/modules

https://registry.terraform.io/

3 Module Types

- Root Modules - You need at least one root module.
- Child Modules - Modules that are called by the root module.
- Published Modules - Modules loaded from a private or public registry.

Calling a Child Module



Scenario Overview

•**Root Module**: This is the entry point. It will use two child modules: one for setting up the VPC and another for deploying an EC2 instance within that VPC.

•**VPC Module (Child Module)**: Creates a VPC with specified CIDR blocks.

•**EC2 Instance Module (Child Module)**: Creates an EC2 instance within the VPC created by the VPC module, applying a security group with specific rules.

```
terraform-project/
|
 ├── main.tf             # Root module
 ├── variables.tf        # Variables definition for root module
 ├── outputs.tf          # Output values from the modules
|
 ├── modules/
    ├── vpc/            # VPC module
   |    ├── main.tf
   |    ├── variables.tf
   |    ├── outputs.tf
   |
   └── ec2/             # EC2 Instance module
        ├── main.tf
        ├── variables.tf
        ├── outputs.tf
```

Root Module

The root module (main.tf) orchestrates the deployment by calling the VPC and EC2 modules, passing necessary variables.

```
# main.tf in the root module

module "vpc" {
  source = "./modules/vpc"
  cidr_block = var.vpc_cidr_block
}

module "ec2_instance" {
  source = "./modules/ec2"
  instance_type = var.instance_type
  vpc_id = module.vpc.vpc_id
}
```

VPC Module (Child Module)
The VPC module creates a Virtual Private Cloud.

```
# main.tf in the VPC module

resource "aws_vpc" "example" {
  cidr_block = var.cidr_block
  tags = {
    Name = "ExampleVPC"
  }
}

# outputs.tf in the VPC module

output "vpc_id" {
  value = aws_vpc.example.id
}
```

Published Modules

```
# Example of using a Third-party Module from the Terraform Registry
module "eks" {
  source          = "terraform-aws-modules/eks/aws"
  version         = "17.1.0"
  cluster_name    = "my-cluster"
  cluster_version = "1.21"
  subnets         = ["subnet-12345678", "subnet-87654321"]
  vpc_id          = "vpc-12345678"
}
```

4 Module Argument Types

- The `source` argument is required for all modules.
- The `version` argument is recommended for modules from a registry.
- The `input variable` arguments.
- The meta-arguments like `for_each` and `depends_on`.

Meta-arguments in Terraform are special arguments available within Terraform resources and modules that can change the behavior of how those resources or modules are handled. They include count, provider, depends_on, lifecycle, and for_each. When used within modules, these meta-arguments can greatly enhance the flexibility and reusability of your Terraform configurations.

Using count to Create Multiple EC2 Instances

In this example, we use count to create a specified number of EC2 instances. This approach is useful when you want to create a fixed number of similar instances, like multiple web servers.

```
variable "instance_count" {
  description = "Number of instances to create"
  type        = number
  default     = 3
}

resource "aws_instance" "example" {
  count         = var.instance_count
  ami           = "ami-0c55b159cbfafe1f0" # Replace this with a valid AMI for your region
  instance_type = "t2.micro"

  tags = {
    Name = "Instance-${count.index}"
  }
}
```

Explanation: This code creates a variable instance_count to define how many EC2 instances you want. The count meta-argument in the aws_instance resource tells Terraform to create that many instances. Each instance will have a unique name tag based on its index (0, 1, 2, etc.).

```
variable "instances" {
  description = "Map of instances with their configurations"
  type = map(object({
    ami           = string
    instance_type = string
  }))
  default = {
    "web" = {
      ami           = "ami-0c55b159cbfafe1f0", # Use appropriate AMI IDs
      instance_type = "t2.micro"
    },
    "app" = {
      ami           = "ami-0c55b159cbfafe1f0",
      instance_type = "t2.small"
    }
  }
}

resource "aws_instance" "example" {
  for_each      = var.instances
  ami           = each.value.ami
  instance_type = each.value.instance_type

  tags = {
    Name = each.key
  }
}
```

Using for_each to Create EC2 Instances with Different Configurations

The for_each example is useful when you have instances that need different configurations, such as varying instance types or roles.

Explanation: This code defines a variable instances as a map, where each key-value pair represents an instance and its configuration. The for_each meta-argument iterates over this map, creating an instance for each entry. Each instance's ami and instance_type are specified according to the map, and the instance is tagged with its key (web, app, etc.).

Summary

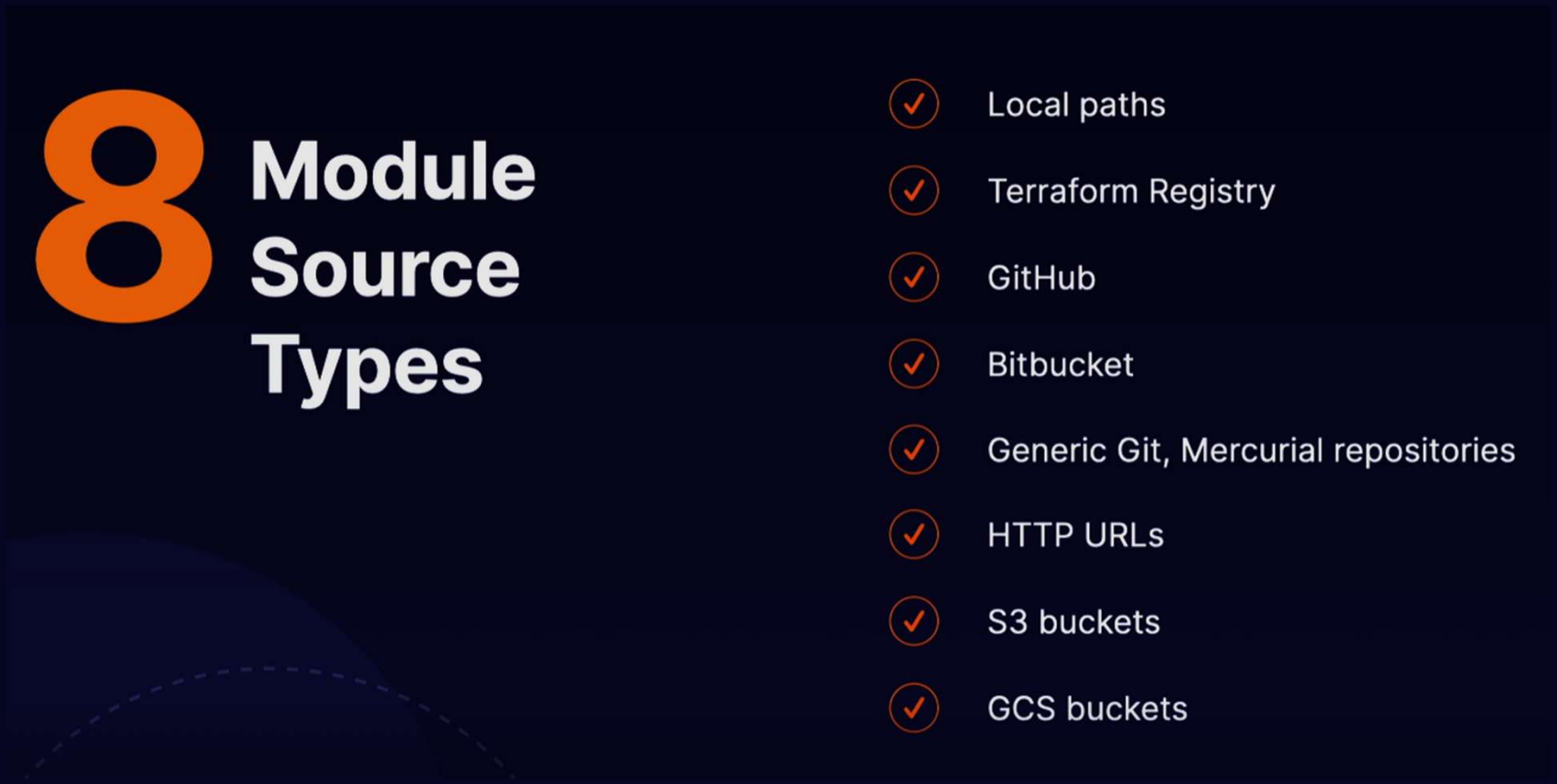
Use count when you need to create a fixed number of similar resources. Use for_each when your resources have different configurations or when you're mapping over a set of values.

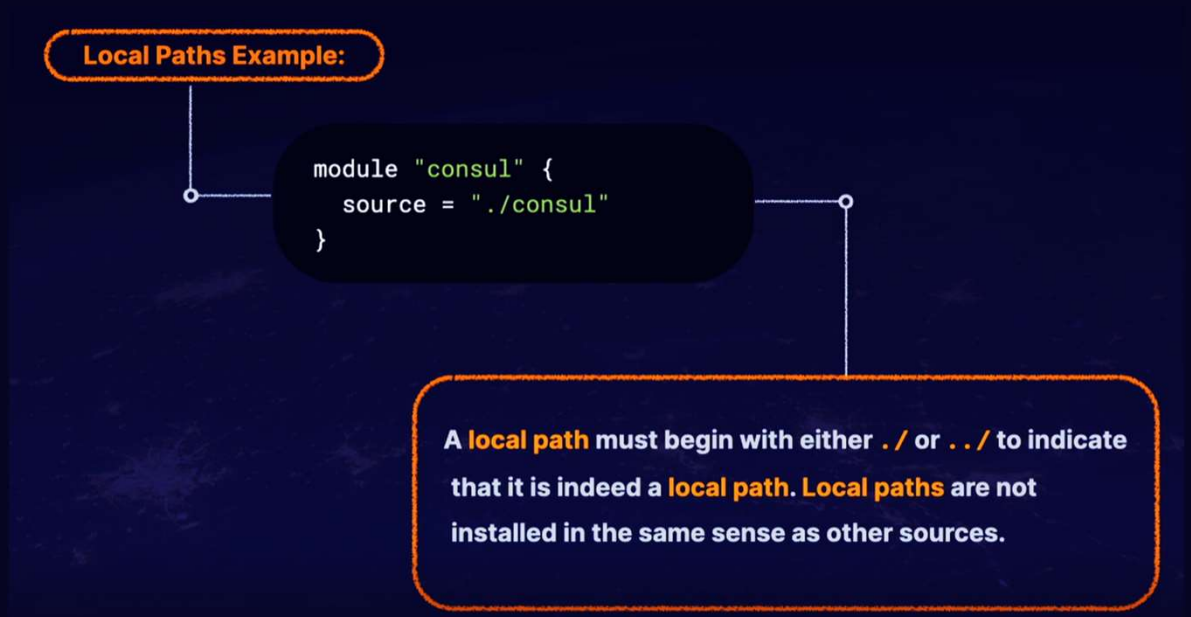
depends_on: Allows you to specify explicit dependencies between modules, ensuring that resources are created or destroyed in a specific order.

provider: Specifies which provider configuration to use for resources within a module, allowing you to use multiple provider instances or configurations.

lifecycle: Inside modules, lifecycle customizations are typically applied to the resources within the module itself, rather than the module block.

Module Sources




Local Paths Example:
module "consul" {
source = "./consul"
}
A local path must begin with either ./ or ../ to indicate that it is indeed a local path. Local paths are not installed in the same sense as other sources.

Using Expressions and Functions

https://developer.hashicorp.com/terraform/language/expressions

https://developer.hashicorp.com/terraform/language/functions

Using Expressions and Functions

The Terraform language uses the following type values:

- `string`
- `number`
- `bool`
- `list/tuple`
- `map/object`
- `null`

Conditional Expressions

Conditional Expression Syntax:

```
condition ? true_val : false_val
```

Result Type Examples:

```
var.example ? 12 : "hello"
```

```
var.example ? tostring(12) :
```

Defining Defaults to Replace Invalid Values:

```
var.a != "" ? var.a : "default-a"
```

In Terraform, the syntax condition ? true_val : false_val is known as the ternary operator. It's a shorthand way to perform an if-else condition.

Basic Understanding
condition: This is a test that evaluates to either true or false.
true_val: If the condition is true, this value is used.
false_val: If the condition is false, this value is used.
It's like asking a yes-or-no question. If the answer is yes (true), you do one thing; if it's no (false), you do another.

var.a != "" ? var.a : "default-a"

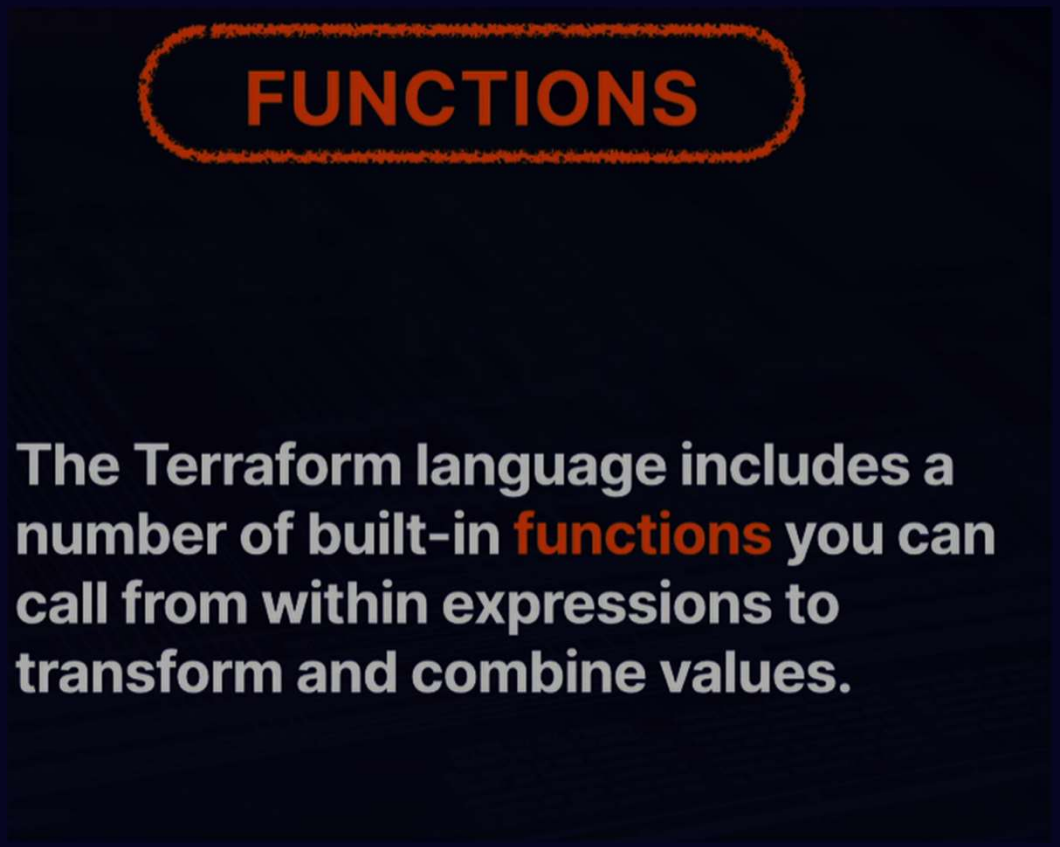
This checks if var.a is not empty (var.a != "").
If var.a is not empty, it uses var.a.
If var.a is empty, it uses "default-a" instead.
Simple Words: If you have a value for a, use it; otherwise, use "default-a".

var.example ? 12 : "hello"

This one is a bit tricky because var.example needs to be a boolean (true or false) for this to make sense. Assuming var.example can be true or false:
If var.example is true, it uses 12.
If var.example is false, it uses "hello".
Simple Words: If example is true, use 12; otherwise, say "hello".

var.example ? tostring(12) : "hello"

Similar to the above, but it explicitly converts 12 to a string using tostring(12). This ensures that whether var.example is true or false, both options are strings.
If var.example is true, it uses "12" (as a string).
If var.example is false, it uses "hello".
Simple Words: If example is true, use "12" (the number as a word); otherwise, say "hello".


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

Built-In Function Syntax:

```
<FUNCTION NAME>(<ARGUMENT 1>, <ARGUMENT 2>)
```

Function Example:

```
min([55, 2453, 2]...)
```

Using Sensitive Data as a Function Argument:

```
> local.baz
{
  "a" = (sensitive)
  "b" = "dog"
}
> keys(local.baz)
(sensitive)
```

Function Call General Syntax:

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

Trying Function Calls Using Terraform Expression Console:

```
terraform console
```

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

```
$ terraform console
> max(5, 12, 9)
12
> exit
cloud_user@jessehoch $
```

Backend Configuration

https://developer.hashicorp.com/terraform/language/settings/backends/configuration

Where They Come From

Terraform includes a built-in selection of backends, and these are the only backends. You cannot load additional backends as plugins.

Where They Are Used

Backend configuration is only used by Terraform CLI. Terraform Cloud and Enterprise always use their own state storage.

What They Do

Two areas of Terraform's behavior are determined by the backend:

- Where state is stored
- Where operations are performed

Backend Block Example:

```
terraform {
  backend "remote" {
    organization = "corp_example"

    workspaces {
      name = "ex-app-prod"
    }
  }
}
```

Backend Block Limitations:

- A configuration can only provide one backend block.
- A backend block cannot refer to named values.

Three Things to Know:

- When the backend changes in a configuration, you must run `terraform init`.
- When the backend changes, Terraform gives you the option to migrate your state.
- HashiCorp recommends you manually backup your state. Simply copy the `terraform.tfstate` file.

local Example Configuration:

```
terraform {
  backend "local" {
    path = "/path/to/terraform.tfstate"
  }
}
```

Configuration Variables:

- `path` - The path to the `tfstate` file.
- `workspace_dir` - The path to non-default workspaces.

Remote Config Block with Workspace:

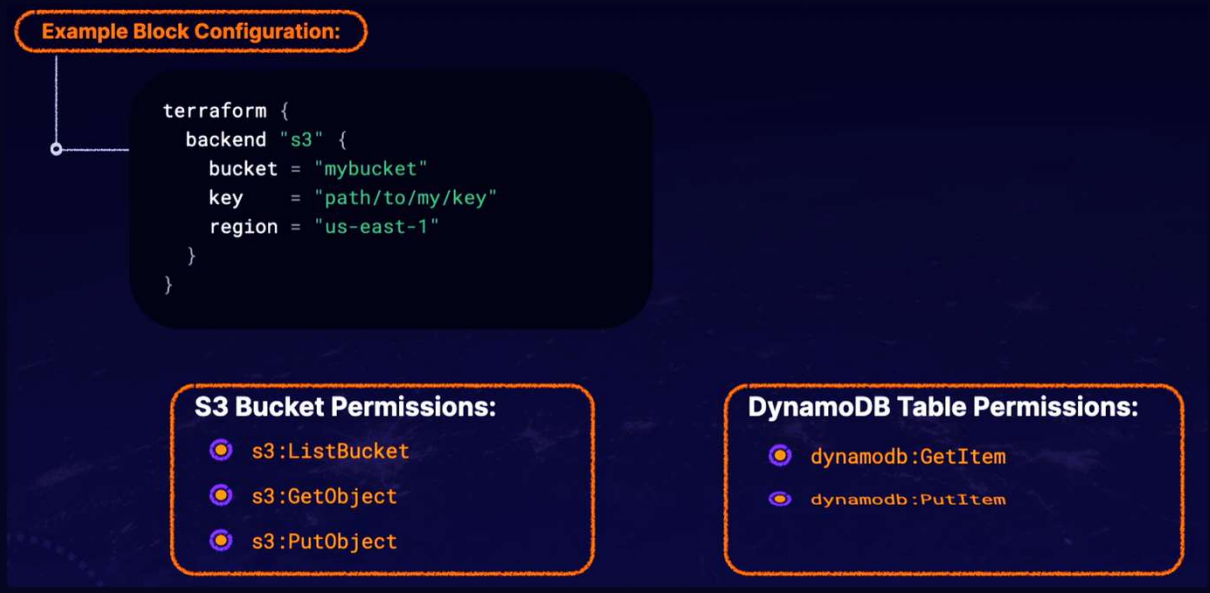
```
terraform {
  backend "remote" {
    hostname = "app.terraform.io"
    organization = "company"

    workspaces {
      name = "my-app-prod"
    }
  }
}
```

Using the CLI Input:

```
# main.tf
terraform {
  required_version = "~> 0.12.0"

  backend "remote" {}
}
```

Backend Configuration

```
terraform {
  backend "s3" {
    bucket         = "my-terraform-state-bucket"  # Replace with your bucket name
    key            = "path/to/my/terraform.tfstate"
    region         = "us-east-1"                  # Replace with your bucket's region
    dynamodb_table = "my-terraform-lock-table"    # Replace with your DynamoDB table name (optional)
    encrypt        = true
  }
}
```



Working with State

https://developer.hashicorp.com/terraform/language/state

Ways Terraform Uses State

Metadata

Terraform must track metadata like resources dependencies. To ensure operation, Terraform retains a copy of the most recent set of dependencies within the state.

Performance

Terraform stores a cache of attribute values for all resources in the state. This comes in handy with large configurations, so you don't have to query all the resources in the configuration.

Syncing

When working in a team, it is recommended to use remote state. It is important that, if working in a team, all members work with the same state. Remote state ensures the state is synced.

State with Backend Configuration

Remote Backend

```
data "terraform_remote_state" "vpc" {
  backend = "remote"

  config = {
    organization = "hashicorp"
    workspaces = {
      name = "vpc-prod"
    }
  }
}

# Terraform >= 0.12
resource "aws_instance" "foo" {
  # ...
  subnet_id =
data.terraform_remote_state.vpc.outputs.subnet_id
}
...
```

Local Backend

```
data "terraform_remote_state" "vpc" {
  backend = "local"

  config = {
    path = "..."
  }
}

# Terraform >= 0.12
resource "aws_instance" "foo" {
  # ...
  subnet_id =
data.terraform_remote_state.vpc.outputs.subnet_id
}
...
```

State Locking

Things to Know:

- If supported by your backend of choice, Terraform locks your state for all operations.
- State locking happens automatically on all operations that could write state.
- Terraform outputs a status message if acquiring the lock is taking longer than expected.
- Not all backends support locking.

Force Unlock Command:

```
$ terraform force-unlock [options] LOCK_ID [DIR]
```

Force Unlock Tips:

- **Be careful with this command!** If you unlock the state when in use, it can cause multiple writes.
- To protect against this, the `force-unlock` command requires a unique lock ID.


$ terraform state <subcommand> [option] [args]
Terraform State Syntax
$ terraform state list
State List
$ terraform state show 'module.name.foo.worker'
State Show
$ terraform state mv packet.foo packet.bar
State Move
$ terraform state rm 'packet.bar'
State Remove
$ terraform state pull
State Pull

Managing Workspaces

https://developer.hashicorp.com/terraform/cli/workspaces

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.

Things to Know:

- Every initialized working directory has at least one **workspace**.
- For a given working directory, only one **workspace** can be selected at a time.
- Most **Terraform** commands only interact with the currently selected **workspaces**.
- Use the `terraform workspace select` command to change the currently selected workspace.
- Use the `terraform workspace list`, `terraform workspace new`, and `terraform workspace delete` commands to manage the available **workspaces** in the current working directory.


workspace list
$ terraform workspace list
default
* development
jsmith-test
workspace select
$ terraform workspace select default
Switched to workspace "default".
workspace show
$ terraform workspace show
development
workspace delete
$ terraform workspace delete example
Deleted workspace "example".

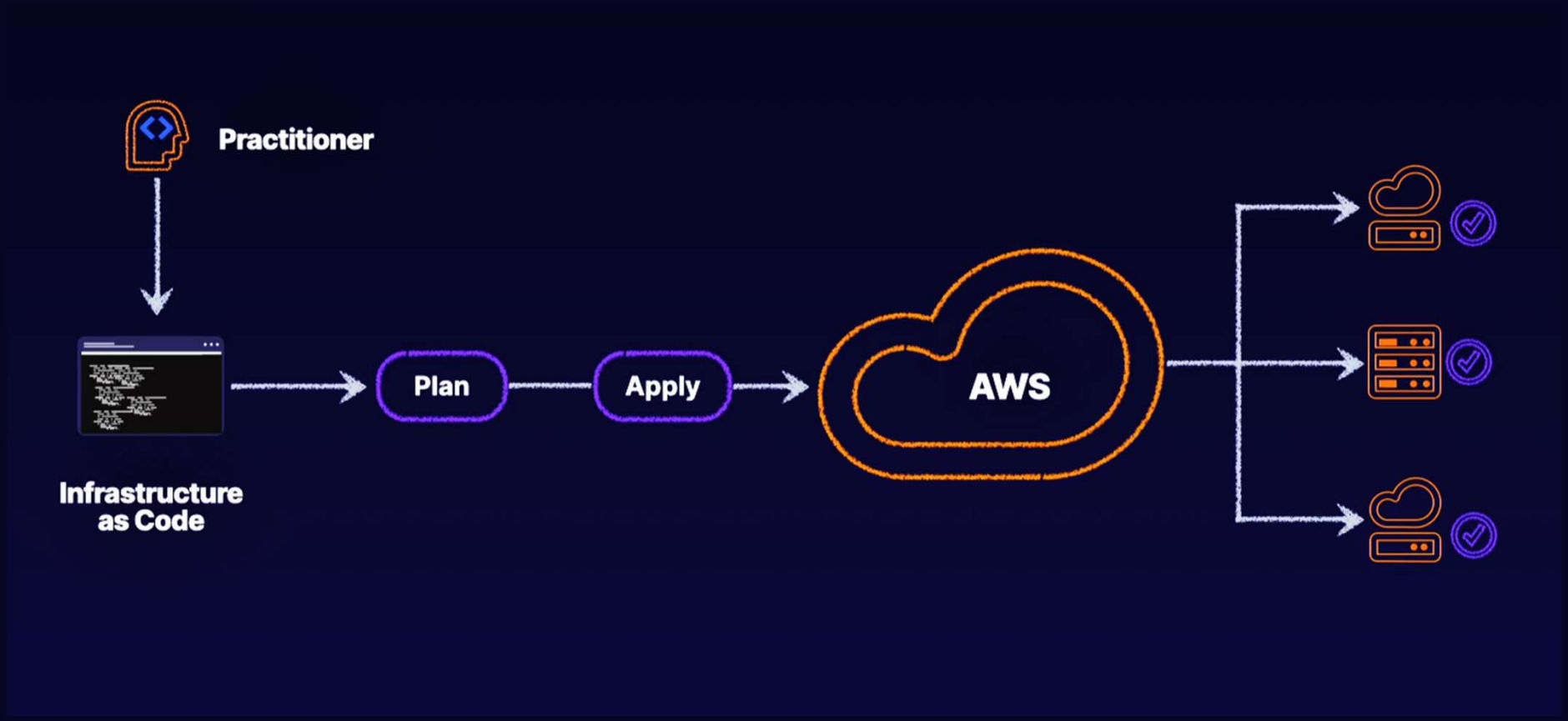
workspace new

```
$ terraform workspace new example
Created and switched to workspace "example"!

You're now on a new, empty workspace. Workspaces isolate their state,
so if you run "terraform plan" Terraform will not see any existing state
for this configuration.
```

workspace new: from state

```
$ terraform workspace new -state=old.terraform.tfstate example
Created and switched to workspace "example".

You're now on a new, empty workspace. Workspaces isolate their state,
so if you run "terraform plan" Terraform will not see any existing state
for this configuration.
```

Using Terraform to Create and Manage Infrastructure on AWS



- Terraform CLI Installed
- AWS CLI Installed

- Authentication
- Building Your Infrastructure
- Making Changes and Destroying Your Infrastructure
- Defining Input Variables
- Using Output Variables to Query Data
- Storing Remote State

The End