Infrastructure-as-Code & Terraform

TEKsystems Global Services

WORKFORCE DEVELOPMENT

Terraform CLI

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

Terraform CLI

Command:

```
terraform graph
```

Output:

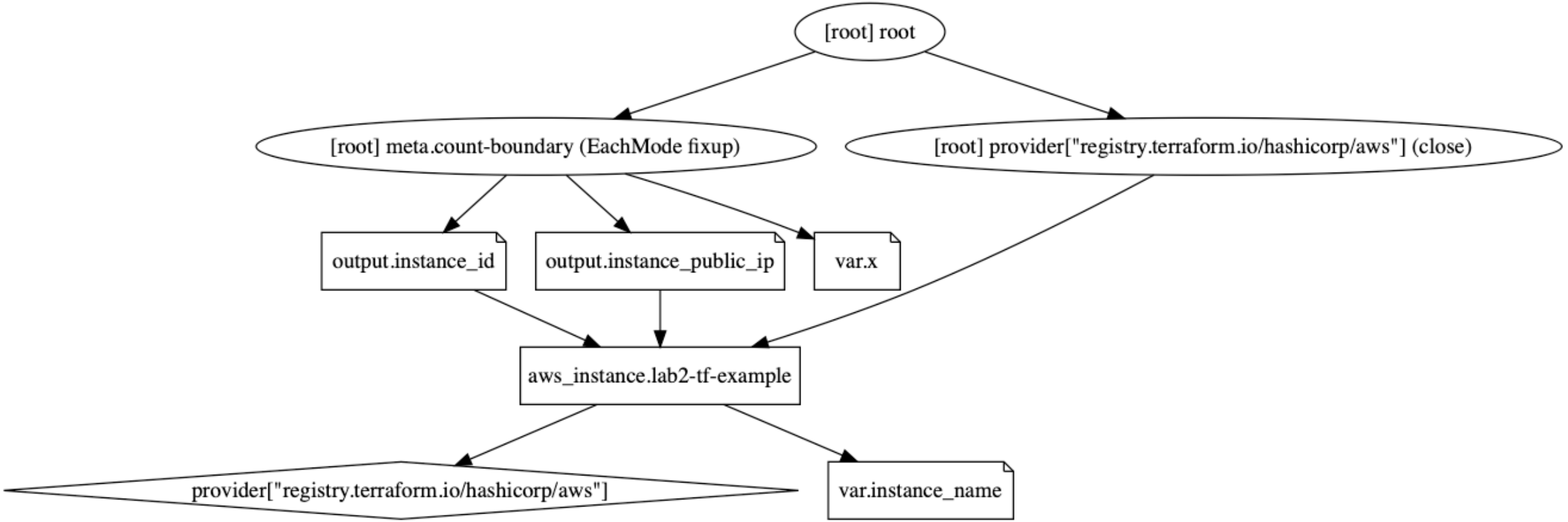
```
digraph {
        compound = "true"
        newrank = "true"
        subgraph "root" {
                "[root] aws_instance.lab2-tf-example (expand)"
[label = "aws_instance.lab2-tf-example", shape = "box"]
...
```

Create a visual graph of Terraform resources

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

Terraform CLI

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



Terraform Resources

Resource Types Overview

```
# Cloud provider resource
resource "aws_instance" "web" {}


# Meta-resource
resource "terraform_remote_state" "network" {}


# Local-only resource
resource "null_resource" "example" {}
```

Resources in Terraform represent different kinds of infrastructure components, each with their own specific configuration patterns.

- Provider-specific resources (aws_instance)
- Meta-resources (terraform_remote_state)
- Null resources for local operations
- Data resources for read-only operations

Terraform resources

Every Terraform resource is structured the exact same way.

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

resource = top level keyword

Resource Arguments

```
resource "aws_instance" "web" {
  # Required arguments
  ami = "ami-0c02fb55956c7d316" # Amazon Linux 2 AMI
in us-east-1
  instance_type = "t2.micro"

  # Meta-arguments
  count = 3

  tags = {
    Name = "web-server"
  }
}
```

Resources accept different types of arguments that define their configuration and behavior.

- Required arguments (name, type)
- Optional arguments with defaults
- Computed arguments (set by provider)
- Meta-arguments (count, for_each)

Lab: Build first instance

Terraform local-only resources

While most resource types correspond to an infrastructure object type that is managed via a remote network API, there are certain specialized resource types that operate only within Terraform itself, calculating some results and saving those results in the state for future use.

For example, you can use local-only resources for:

- Generating private keys
- Issue self-signed TLS certs
- Generating random ids

- The behavior of local-only resources is the same as all other resources, but their result data exists only within the Terraform state. "Destroying" such a resource means only to remove it from the state, discarding its data.

Terraform local-only resources

Local-only resources are also referred to as logical resources

This is primarily used for easy bootstrapping of throwaway development environments.

```
resource "tls_private_key" "example" {
  algorithm   = "ECDSA"
  ecdsa_curve = "P384"
}
```

Local-Only Resources - Random

```
resource "random_id" "bucket_suffix" {
  byte_length = 8
}


resource "random_password" "db_password" {
  length = 16
  special = true
  override_special = "!#$%&*()-_=+[]{}<>:?"
}
```

Random resources generate and maintain consistent random values
across Terraform runs.

- Random strings
- Random integers
- UUIDs
- Passwords

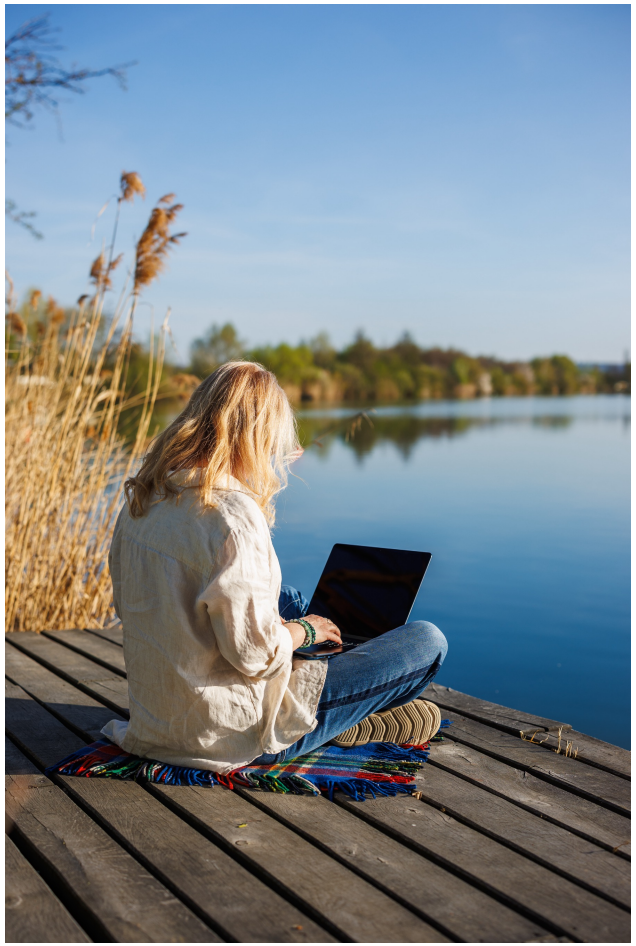
Local-Only Resources - Time

```
resource "time_rotating" "key_rotation" {
  rotation_days = 30
}


resource "aws_kms_key" "example" {
  description = "crypto-key"
  rotation_enabled = true
}
```

Time resources help manage time-based operations and dependencies.

- Rotation triggers
- Delayed operations
- Time-based dependencies
- Schedule tracking

Understanding depends_on



Terraform automatically builds a dependency graph from your references, but some resources do not have a direct link. In those cases you can use depends_on to tell Terraform the correct order.

- Terraform normally figures out dependencies on its own
- Use depends_on only when the relationship is not obvious
- Ensures a resource waits for another resource to finish
- Helpful for things like IAM policies, permissions, or order-sensitive setups

Use it sparingly. Terraform prefers automatic dependencies, but depends_on protects you when the order truly matters.

Depends on Example

```
resource "aws_iam_role_policy_attachment" "attach" {
  role       = aws_iam_role.app.name
  policy_arn = aws_iam_policy.app.name

  depends_on = [aws_iam_role_policy.app_permissions]

}
```

role is the IAM role that will receive the permission

policy_arn is the ARN of the policy we want to attach

depends_on guarantees the **app_permissions** policy exists first

Terraform Lifecycle

The lifecycle block lets you control how Terraform creates, updates, and destroys resources. It gives you extra control when the default behavior is not enough, especially when dealing with sensitive or order-dependent resources.

- Can prevent Terraform from destroying important resources
- Can force Terraform to create replacements before deleting old ones
- Can tell Terraform to ignore certain changes in the cloud

Lifecycle is used to protect resources and guide Terraform's behavior during changes.

Common Uses for Lifecycle

Lifecycle settings help you handle situations where Terraform's normal behavior might break something, cause downtime, or overwrite values that should be ignored. These options give you extra safety and control.

- Preventing Terraform from destroying critical resources
- Requiring Terraform to create a new resource before removing the old one
- Ignoring fields that change outside Terraform, such as tags or timestamps

Lifecycle is ideal when resources are sensitive, shared, or managed by multiple systems.

Lifecycle Example

```
resource "aws_s3_bucket" "logs" {
 bucket      = "app-logs-prod"

   lifecycle {
   prevent_destroy         = true
   ignore_changes          = [tags]
   create_before_destroy   = true
  }

}
```

This lifecycle block protects the bucket, ignores tag changes made outside Terraform, and ensures replacements happen safely without deleting the original first.

Terraform provisioners

Terraform is designed to programmatically create and manage infrastructure. It is not intended to replace Ansible, Chef or any other configuration management tool.

It does include provisioners as a way to prepare the system for your services.

Generic provisioners:

- file
- local-exec
- remote-exec

Provisioner Overview

```
resource "google_compute_instance" "web" {
  name = "web-server"
  machine_type = "e2-micro"


provisioner "remote-exec" {
  inline = [
    "apt-get update",
    "apt-get install -y nginx"
    ]
  }
}
```

Provisioners allow you to execute actions on local or remote machines as part of resource creation or destruction.

- Post-creation configuration
- Software installation
- File transfers
- Command execution

Terraform provisioners

Generic Provisioners:

- file
  - The file provisioner is used to copy files or directories from the machine executing Terraform to the newly created resource. The file provisioner supports both SSH and WinRM type connections.
- local-exec
  - The local-exec provisioner invokes a local executable after a resource is created. This invokes a process on the machine running Terraform, not on the resource.
- remote-exec
  - The remote-exec provisioner invokes a script on a remote resource after it is created. This can be used to run a configuration management tool, bootstrap into a cluster, etc. The remote-exec provisioner supports both SSH and WinRM type connections.

Remote-Exec Provisioner

```
resource "google_compute_instance" "web" {
  name = "web-server"

  connection {
    type = "ssh"
    user = "admin"
    private_key =
file("${path.module}/ssh/private_key")
  }

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

Remote-exec provisioners run commands on the remote resource after creation.

- Install software packages
- Configure services
- Start applications
- Run setup scripts

Local-Exec Provisioner

```
resource "google_compute_instance" "db" {
  name = "database"


  provisioner "local-exec" {
    command = <<-EOT
    echo
"DB_HOST=${self.network_interface[0].network_ip}" >>
.env
    ./scripts/update-dns.sh ${self.name}
${self.network_interface[0].access_config[0].nat_ip}
    EOT
  }
}
```

Local-exec provisioners run commands on the machine executing Terraform.

- Generate configuration files
- Run local scripts
- Trigger external tools
- Update documentation

File Provisioner

```
resource "google_compute_instance" "app" {
  name = "application"


  connection {
    type = "ssh"
    user = "admin"
    private_key = file(var.ssh_private_key)
  }

  provisioner "file" {
    source = "configs/"
    destination = "/etc/application/"
  }
}
```

File provisioners copy files or directories to the remote resource.

- Configuration files
- Scripts
- Application code
- SSL certificates

Lab: Build first instance

Terraform variables

Input variables serve as parameters for a Terraform module, allowing aspects of the module to be customized without altering the module's own source code, and allowing modules to be shared between different configurations.

When you declare variables in the root module of your configuration, you can set their values using CLI options and environment variables. When you declare them in child modules, the calling module should pass values in the module block.

Terraform variables

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

The label after the variable keyword is a name for the variable, which must be unique among all variables in the same module. This name is used to assign a value to the variable from outside and to reference the variable's value from within the module.

```
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"
    }
  ]
}
```

Terraform variables

Terraform has some reserved variables:

- source
- version
- providers
- count
- for_each
- lifecycle
- depends_on
- locals

- These are reserved for meta-arguments in module blocks.

```
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"
    }
  ]
}
```

Terraform variables

Optional variable arguments:

- default: A default value which makes the variable optional
- type: Specifies value type accepted for the variable.
- description: Specifies the variable's documentation
- validation: A block to define validation rules
- sensitive: Does not show value in output

Environment-Specific Variable Files

```
# dev.tfvars
environment = "dev"
project_id = "my-project-dev"
node_count = 1
instance_type = "t3.medium"


# prod.tfvars
environment = "prod"
project_id = "my-project-prod"
node_count = 3
instance_type = "t3.large"
```

Using separate tfvars files allows managing multiple environments with the same code base.

- One configuration, multiple environments
- Clear separation of config from code
- Easy environment promotion
- Consistent resource naming

Local Value Patterns

```
locals {
  resource_prefix = "${var.environment}-
${var.project_id}"
  common_tags = {
    Environment = var.environment
    Project = var.project_id
    ManagedBy = "terraform"
  }
  instance_name = "${local.resource_prefix}-
instance"
  bucket_name = "${local.resource_prefix}-
storage"
}
```

Local values help create consistent naming and tagging conventions across resources.

- Combine multiple variables
- Create reusable patterns
- Enforce naming conventions
- Reduce repetition

Output Management

```
output "environment_info" {
  value = {
    name = var.environment


    resources = {
      instances = aws_instance.web[*].tags["Name"]
      bucket = aws_s3_bucket.data.bucket
      vpc = aws_vpc.main.tags["Name"]
  }

  endpoints = {
    api = "https://${local.resource_prefix}-
api.example.com"
    web = "https://${local.resource_prefix}-
web.example.com"
    }
  }
}
```

Outputs expose environment-specific values while maintaining consistency across environments.

- Environment-specific values
- Consistent output structure
- Cross-environment references
- Pipeline integration points

Environment-Specific Conditions

```
locals {
  is_production = var.environment == "prod"
  instance_count = {
    dev = 1
    staging = 2
    prod = 3
  }
}
```

Use variables to create environment-specific resource configurations.

- Conditional resource creation
- Environment-based sizing
- Feature flags
- Resource counts

Terraform Workspaces

Workspaces let you keep multiple state files inside the same Terraform configuration. Each workspace represents a separate environment, so you can reuse the same code without mixing resources.

- Default workspace is named default
- Each workspace has its own state file
- Useful for dev, test, and staging environments
- Keeps environments isolated while using the same Terraform code
- Workspaces = same code, separate states.

Terraform CLI

Command:

```
terraform workspace
```

Output:

```
Usage: terraform [global options] workspace

  new, list, show, select and delete Terraform workspaces.

Subcommands:
    delete    Delete a workspace
    list      List Workspaces
    new       Create a new workspace
    select    Select a workspace
    show      Show the name of the current workspace
```

The usage instructions shows all subcommands for workspaces

Terraform CLI

Command:

```
terraform workspace list
```

Output:

```
default
dev
* prod
```

The * indicates the active environment

Environment-Specific Variable Files

```
# dev.tfvars
environment = "dev"
project_id = "my-project-dev"
node_count = 1
instance_type = "t3.medium"


# prod.tfvars
environment = "prod"
project_id = "my-project-prod"
node_count = 3
instance_type = "t3.large"
```

Using separate tfvars files allows managing multiple environments with the same code base.

- One configuration, multiple environments
- Clear separation of config from code
- Easy environment promotion
- Consistent resource naming

Terraform CLI

Prod Workspace:

```
terraform workspace new prod
Terraform workspace select prod
terraform apply -var-file="prod.tfvars"
```

Dev Workspace:

```
terraform workspace new dev
Terraform workspace select dev
terraform apply -var-file="dev.tfvars"
```

Each workspace has its own state that it tracks. The variable values are set from the respectice tfvars file.

Lab: Workspaces tfvars

Terraform Functions

Terraform includes built-in functions that help you transform data, combine values, and make your configurations more flexible. They do not create resources. They only operate on values inside your code.

- Formatting strings, merging maps, and working with lists
- Help avoid hard-coding values
- Make modules more reusable
- Common in variables, locals, and resource arguments

Functions give Terraform configurations more power without adding complexity.

Terraform CLI

Command:

```
terraform console
```

Output:

```
> var.x[1].name
"second"

var.x[0].condition
{
  "age" = "1"
}
```

Test expressions and interpolations interactively.

Terraform CLI

Command:

```
terraform console -var-file=dev.tfvars
```

Output:

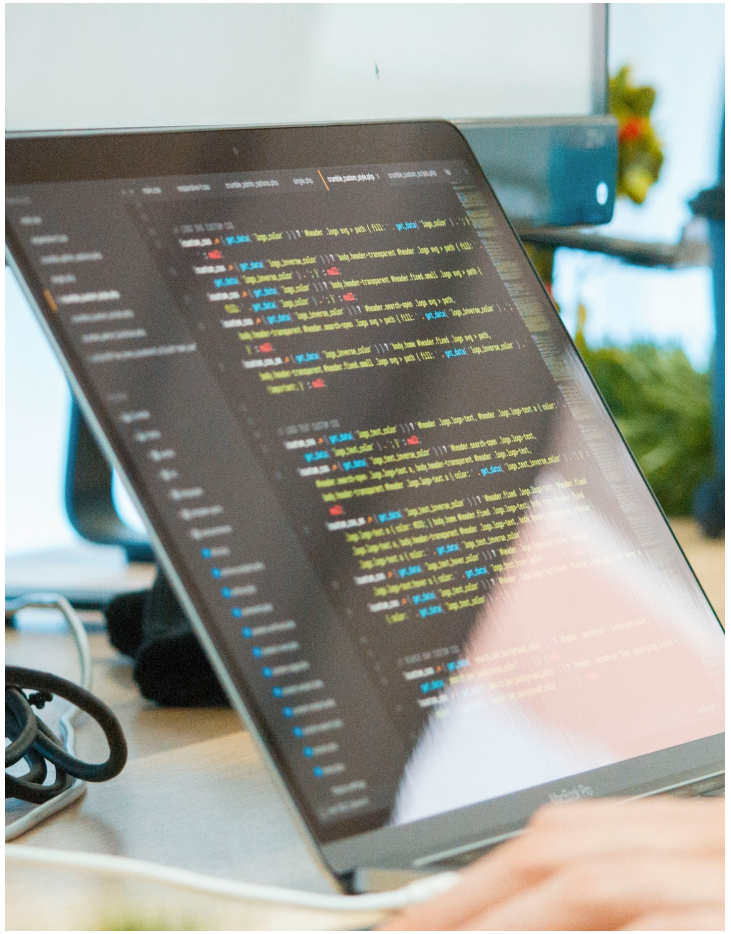
```
> upper(var.instance_type)
"T2.MICRO"

> length(var.instance_type)
8
```

Terraform console can load from a tfvars file

Using Functions in HCL

HCL does not support creating your own custom functions. Terraform only provides built-in functions, and modules are the correct way to reuse logic. Functions simply help you process data before passing it into resources or modules.

Common built-in Functions



- `format()` – build strings dynamically
- `join()` – combine list elements
- `split()` – break strings into lists
- `merge()` – merge maps
- `length()` – count items
- `contains()` – check for a value
- `lower()` / `upper()` – clean up text
- `lookup()` – safe map lookup

Functions prepare values. Modules reuse infrastructure logic.

Environment-Specific Variable Files

```
locals {
  server_name = format("%s-%s-%s",
    var.env, var.app, "server")
}

resource "aws_s3_bucket" "logs" {
  bucket = lower(local.server_name)
}
```

This example uses `format()` to build a predictable name and `lower()` to ensure it meets AWS naming rules. Functions help generate clean values before passing them into resources.

Lab: Workspaces tfvars

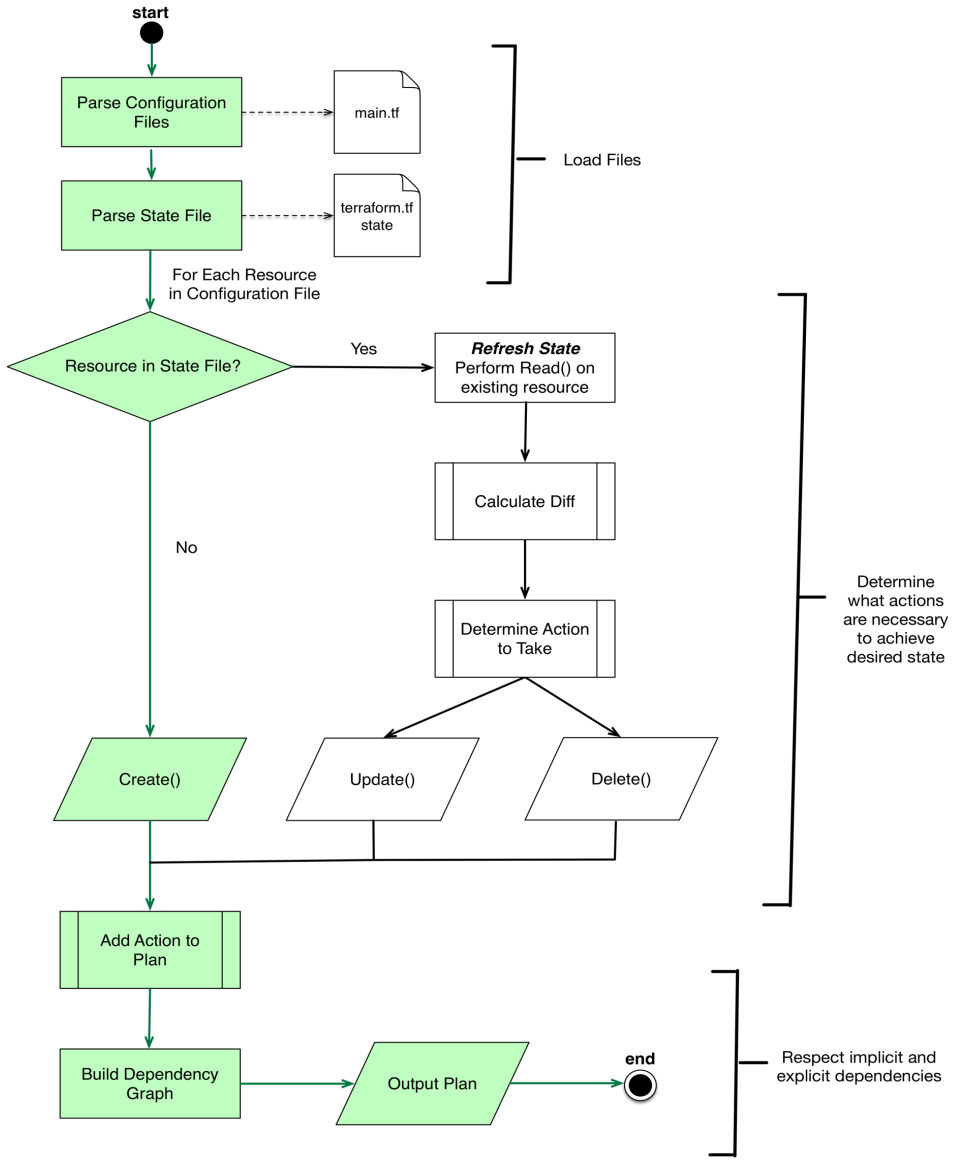
Terraform State Management

When Terraform creates a new infrastructure object represented by a resource block, the identifier for that real object is saved in Terraform's state, allowing it to be updated and destroyed in response to future changes.

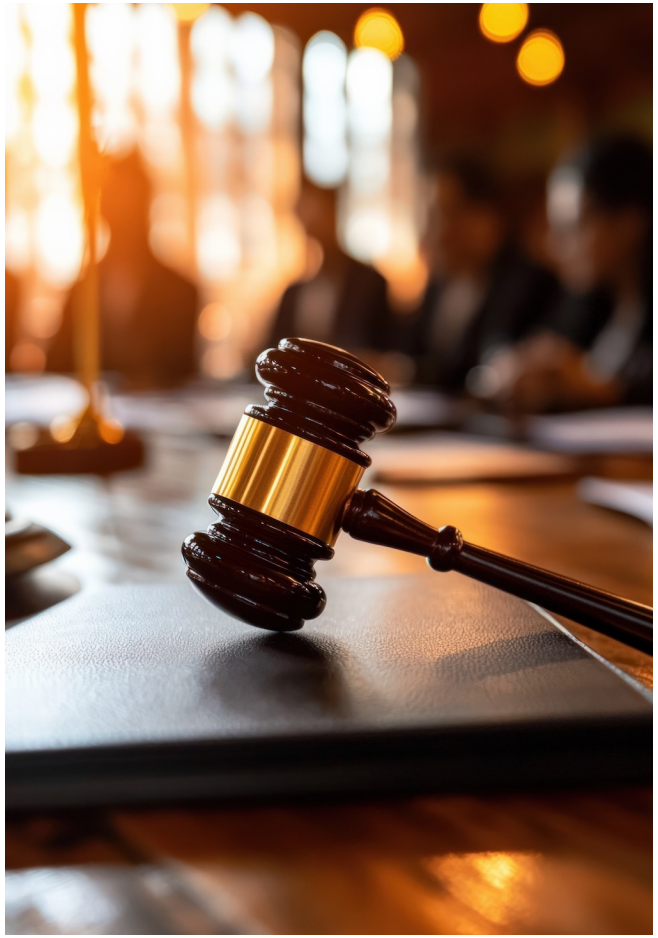
Terraform State Management

Resource blocks that already have an associated infrastructure object in the state, Terraform compares the actual configuration of the object with the arguments given in the configuration and, if necessary, updates the object to match the configuration.

Terraform State Management



Terraform State Management



State is Terraform's record of your real infrastructure. Managing state correctly ensures Terraform knows what exists, what needs updating, and what should not be changed. Good state management prevents drift, errors, and accidental deletions.

- State tracks every resource Terraform creates
- Remote backends keep state safe and shared
- Locking prevents conflicting updates
- Import lets Terraform manage existing resources
- Removing items from state makes Terraform ignore them

Proper state management keeps your infrastructure predictable and stable.

Infrastructure Drift Prevention

Drift prevention is the practice of implementing controls and processes to stop infrastructure changes from happening outside of your IaC tool. It involves setting up guardrails, access controls, and automated monitoring to ensure all infrastructure changes go through your approved Terraform workflows.

- Comprehensive tagging policies - Ensure all resources have consistent tags
- Resource locking - Prevent manual modifications to critical resources
- Regular monitoring - Set up alerts for unexpected changes
- Access controls - Limit who can modify infrastructure Outside of Terraform

Adjusting State When Needed

Although not ideal, some situations require manually adding or removing resources from Terraform state. These actions help prevent accidental recreation or destruction when Terraform's view does not match reality.

- Import an existing resource so Terraform manages it
- Remove a resource from state before destroying it safely
- Useful for sensitive items like S3 buckets or IAM roles
- Prevents Terraform from recreating or deleting critical resources

State adjustments keep your environment safe when Terraform cannot handle the change automatically.