**Infrastructure Automation: Top Terraform Hacks**

As we know with Terraform, you can declaratively define and provision your infrastructure across various cloud and on-prem providers.

Based on my previous guide explored in past, I will go over some useful Terraform hacks and techniques with practical examples to help streamline your infrastructure automation.

**Overview**

I will cover the following Terraform automation hacks:

* **Modules** — Reusable, encapsulated Terraform configs for infrastructure components
* **Workspaces** — Multiple isolated working states for organizations and environments
* **Remote State** — Storing state in remote backend for team access
* **Data Sources** — Looking up existing resources and using their attributes
* **Splat Expressions** — Referencing resources dynamically using splat expressions
* **for\_each** — Provisioning multiple similar resources from a map/set
* **Providers** — Extending Terraform through custom providers
* **null\_resource** — Running provisioners and local execs
* **Templates** — Generating custom config files from templates
* **Private Module Registry** — Hosting and sharing private modules

Let’s look at each of these hacks in more detail with examples.

**Modules**

Terraform modules allow you to define reusable components so you can avoid repeating the same resource configuration. Modules encapsulate infrastructure into a single unit with inputs and outputs.

For example, we can define a reusable module for a Kubernetes cluster:

# k8s-cluster module  
variable "name" {}  
variable "location" {}  
variable "node\_count" {}

resource "azurerm\_resource\_group" "this" {  
 name = "${var.name}"  
 location = "${var.location}"  
}resource "azurerm\_kubernetes\_cluster" "this" {  
 name = "${var.name}"  
 location = "${var.location}"  
 resource\_group\_name = azurerm\_resource\_group.this.name  
 dns\_prefix = "${var.name}" default\_node\_pool {  
 name = "default"  
 node\_count = "${var.node\_count}"  
 vm\_size = "Standard\_D2\_v2"  
 }  
}output "client\_key" {  
 value = azurerm\_kubernetes\_cluster.this.kube\_config.0.client\_key  
}output "client\_certificate" {  
 value = azurerm\_kubernetes\_cluster.this.kube\_config.0.client\_certificate  
}

We can then use this module in our main config to create a reusable Kubernetes cluster component:

# main.tf  
module "k8s" {  
 source = "./modules/k8s-cluster"  
 name = "demo-cluster"  
 location = "eastus"  
 node\_count = 3  
}

This allows us to encapsulate infrastructure into modular, composable units that can be invoked just like functions.

Some best practices when using Terraform modules:

* Put shared modules in their own Git repo for easy reuse
* Make modules configurable via input variables
* Minimize module dependencies for loose coupling
* Only expose necessary values via outputs
* Give modules high cohesion, low coupling
* Version modules using Git tags for change control

Well-structured modules help you avoid repetition, enforce standards, and simplify configs.

**Workspaces**

Terraform workspaces allow having multiple isolated working states for the same configuration. This lets you have different resources for dev, staging, production, etc.

To use workspaces, first initialize Terraform with workspaces enabled:

terraform init -backend-config=workspaces.enabled=true

You can then create and switch workspaces:

# Create workspaces  
terraform workspace new staging  
terraform workspace new production

# Switch workspaces  
terraform workspace select staging

Now any resources you create will apply to the workspace you selected. You can have totally different resources, accounts, providers per workspace.

Some examples where workspaces shine:

* **Multi-environment** — Create separate dev, test, prod environments from same config
* **Isolation** — Safely try changes in new workspaces without affecting others
* **Organization** — Separate workspaces per department, project, microservice

Workspaces allow working on infrastructure components in parallel without crosstalk between environments and teams.

**Remote State**

By default, Terraform stores state locally in a file. For team usage, you instead want to store state remotely using a backend like S3.

Here is an example configuring S3 as a remote state backend:

terraform {  
 backend "s3" {  
 bucket = "terraform-state-prod"   
 key = "main/terraform.tfstate"  
 region = "us-east-1"  
 }  
}

This stores state centrally so any team member can access it. Remote state also enables:

* **State locking** — Prevents race conditions with concurrent runs
* **Access controls** — Restrict state access through ACLs and encryption
* **Change history** — Ability to revert state and see historical changes

Other common state backend options include Azure Storage, GCP Storage, Terraform Cloud, Hashicorp Consul.

For larger teams and organizations, remote state is essential for safe collaboration and change management.

**Data Sources**

Terraform data sources allow you to fetch information about existing resources outside of Terraform. This lets you avoid managing the resource directly and reference its attributes.

For example, you can lookup an existing Kubernetes cluster and use its credentials:

# Lookup existing K8s cluster  
data "google\_container\_cluster" "my\_cluster" {  
 name = "existing-cluster"  
 zone = "us-central1-a"  
}

# Use cluster credentials   
provider "kubernetes" {  
 host = data.google\_container\_cluster.my\_cluster.endpoint  
 cluster\_ca\_certificate = base64decode(data.google\_container\_cluster.my\_cluster.master\_auth.0.cluster\_ca\_certificate)  
 token = data.google\_container\_cluster.my\_cluster.master\_auth.0.client\_token  
}

You can also query data from the providers themselves, like available VM sizes for an Azure region:

data "azurerm\_resource\_group" "rg" {  
 name = "existing-rg"  
}

data "azurem\_virtual\_machine\_sizes" "sizes" {  
 location = data.azurerm\_resource\_group.rg.location  
}output "vm\_sizes" {  
 value = data.azurem\_virtual\_machine\_sizes.sizes.options  
}

Data sources allow accessing existing data without managing the lifecycle of the underlying resource.

**Splat Expressions**

Splat expressions provide a compact way to reference resources dynamically in Terraform.

The \* syntax lets you efficiently query resource attributes and interpolate these into other configs.

For example, to dynamically loop through subnets:

resource "aws\_subnet" "example" {  
 count = 3  
   
 vpc\_id = aws\_vpc.main.id  
 cidr\_block = "10.0.${count.index}.0/24"  
 availability\_zone = data.aws\_availability\_zones.available.names[count.index]   
}

resource "aws\_directory\_service\_directory" "example" {  
 vpc\_settings {  
 subnet\_ids = aws\_subnet.example[\*].id   
 }  
}

The aws\_subnet.example[\*].id expression extracts the id attribute from each subnet in the example set into a list that can be referenced elsewhere.

Splat expressions are useful for:

* Dynamically extracting attributes from resources
* Referencing maps, lists from inputs and outputs
* Iterating over complex nested data structures

By using splat expressions, you can reduce boilerplate and make configurations more compact and readable.

**for\_each**

The for\_each meta-argument provides a way to dynamically generate multiple instances of a resource from a map, set, or lookup table. This avoids manually specifying repetitive resource blocks.

For example, you can generate multiple subnets from a set:

variable "subnet\_zones" {  
 default = ["us-east-1a", "us-east-1b", "us-east-1c"]  
}

resource "aws\_subnet" "example" {  
 for\_each = toset(var.subnet\_zones) vpc\_id = aws\_vpc.main.id  
 cidr\_block = cidrsubnet(aws\_vpc.main.cidr\_block, 4, each.key)   
 availability\_zone = each.value  
}

You can also generate resources from a map:

variable "subnets" {  
 default = {  
 zone1 = "10.0.1.0/24"  
 zone2 = "10.0.2.0/24"  
 }  
}

resource "aws\_subnet" "example" {  
 for\_each = var.subnets vpc\_id = aws\_vpc.main.id  
 cidr\_block = each.value  
 availability\_zone = each.key  
}

The for\_each meta-argument is very flexible and allows you to minimize repetition in your configs.

**Providers**

Terraform providers enable extending Terraform to manage resources from new APIs.

Custom providers can be written in Go by implementing the Terraform plugin SDK. For example, you could build a provider for a SaaS application.

Providers expose resources and data sources for the APIs they wrap. They run as plugins that Terraform executes during runs.

Here is an example outline for a custom Terraform provider:

// Implement Provider interface  
type Provider struct { }

// Provider function  
func Provider() \*schema.Provider {  
 // Return provider schema  
}// Define provider resources  
func resourceExample() \*schema.Resource {  
 // Return schema for a new resource  
}// Define provider data sources   
func dataSourceExample() \*schema.Resource {  
 // Return schema for a new data source  
}// Implement resource and data source CRUD handlers  
func resourceExampleCreate(d \*schema.ResourceData, m interface{}) error {  
 // Create logic goes here  
}

To build and distribute a custom provider, you can use the Terraform Plugin Framework and publish it on the Terraform Registry.

Custom providers are powerful for extending Terraform to non-native tools and APIs.

**null\_resource**

The null\_resource is a handy way to run provisioners and local execs as part of your Terraform workflow.

A null\_resource has no direct infrastructure impact but can execute scripts to install software, run configuration management, etc.

For example:

resource "null\_resource" "configure\_server" {  
 provisioner "local-exec" {  
 command = "ansible-playbook playbook.yml"  
 }  
   
 triggers = {  
 always\_run = timestamp()  
 }  
}

The triggers define when the provisioner will run, which you can base on values changing.

null\_resource is useful for:

* Running ad-hoc scripts without a dedicated resource
* Executing provision steps on create/destroy
* Implementing orchestration and coordination
* Running configuration management like Ansible

By leveraging null\_resource, you can extend Terraform to include external tools in the workflow.

**Templates**

Along with provisioners, templates provide another way to generate custom config files with Terraform.

The templatefile function renders text from a template, injecting values from variables.

For example:

# nginx.conf.tpl  
server {  
 listen ${port}  
 server\_name ${domain}  
 root ${webroot}  
}

# main.tf  
variable "port" {}  
variable "domain" {}resource "local\_file" "nginx\_config" {  
 filename = "/tmp/nginx.conf" content = templatefile("${path.module}/nginx.conf.tpl", {  
 port = var.port  
 domain = var.domain   
 webroot = "/var/www/html"  
 })  
}

Here we generate an Nginx config file by interpolating values into the template.

You can also use templating with cloudinit\_config for EC2 user data:

data "template\_cloudinit\_config" "config" {  
 gzip = true  
 base64\_encode = true

part {  
 filename = "init.cfg"  
 content\_type = "text/cloud-config"  
 content = templatefile("${path.module}/init.cfg.tpl", {  
 foo = "bar"  
 })  
 }  
}resource "aws\_instance" "example" {  
 # ... user\_data = data.template\_cloudinit\_config.config.rendered  
}

Templates allow separating infrastructure definition from configuration details.

**Private Module Registry**

Terraform has a public module registry at registry.terraform.io containing modules you can reuse.

For private modules, you can also run your own module registry using the open source Terraform Registry or tools like Artifactory.

This lets you publish modules privately within your organization:

To configure a private registry:

terraform {  
 backend "s3" {...}

module\_sources {  
 private = "https://private.terraform.dso.io"   
 }  
}

You can then reference private modules:

module "vpc" {  
 source = "private/vpc/aws"  
}

This provides a searchable, centralized catalog of approved modules conforming to standards. Private registries are great for large teams and organizations.

**Conclusion**

These Terraform tips help simplify infrastructure automation:

* **Modules** — Encapsulate reusable components
* **Workspaces** — Isolate states for different environments
* **Remote State** — Store state centrally for team access
* **Data Sources** — Lookup existing resource attributes
* **Splat** — Dynamically extract values into expressions
* **for\_each** — Iterate over maps and sets for resources
* **Providers** — Extend Terraform through custom providers
* **null\_resource** — Execute scripts, provisioners, and configuration management
* **Templates** — Generate custom configs and user data
* **Private Registry** — Share approved modules internally

Terraform is a versatile tool for codifying infrastructure. Mastering these hacks will help you use Terraform more effectively to automate provisioning, changes, and orchestration. You can build robust, reusable infrastructure components to streamline delivering infrastructure at scale.