Module 08
Terraform – Other Topics

Contents

- Terraform Provisioners
- Terraform Workspaces
- Other topics

Terraform Provisioners

Important: Use provisioners as a last resort. There are better alternatives for most situations. Refer to Declaring Provisioners for more details.

Terraform Provisioners - Intro

- *Definition* from Terraform: "You can use provisioners to model specific actions on the local machine or on a remote machine in order to prepare servers or other infrastructure objects for service."
- In simple terms
  - You use Terraform to create/provision servers (VMs), etc…
  - You use provisioners to configure those servers (VMs) – install software, etc.
- Two types of provisioners:
  - Local – configures something in the machine were you are running Terraform
  - Remote – configure something in a remote machine (typically one or more VMs you just created with terraform) - Requires credentials!!
- Terraform Docs
- Terraform explicitly states that Provisioners are a last resource –
  - Use only if strictly necessary (seldom)
  - Most clouds give you some built in tools to configure VMs: e.g. user_data in AWS, custom_data in Azure, metadata in GCP.
    - These built-in mechanisms may suffice in many cases (although it is often better to bake "Golden AMIs")

Important: Use provisioners as a last resort. There are better alternatives for most situations. Refer to Declaring Provisioners for more details.

local-exec Provisioner

- Two types:
  - Creation-time (default)
  - Destroy-time
- Creation-time provisioners are only run during *creation*, not during updating or any other lifecycle
- Multiple provisioners can be specified within a resource. They are executed in the order they're defined in the configuration file.
- "self" is used to refer to the parent object (in this example aws_instance.server)
- Example in ”lab_10_provisioners”

```
resource "aws_instance" "server" {
  count         = var.num_instances
(...)

  ## NOTE : HashiCorp discourages the use of provisioners!
  ## https://developer.hashicorp.com/terraform/language/resources/provisioners/syntax#provisioners-are-a-last-resort
  provisioner "local-exec" {
    when = create
    command = "mkdir -p ${path.root}/temp"
  }
  provisioner "local-exec" {
    when = create
    command = "echo Server ${count.index}: IP address is ${self.private_ip} >> ${path.root}/temp/tempfile.txt"
  }
  provisioner "local-exec" {
    when = destroy
    command = "rm -rf ${path.root}/temp"
  }
}
```

Important: Use provisioners as a last resort. There are better alternatives for most situations. Refer to Declaring Provisioners for more details.

file and remote-exec Provisioners

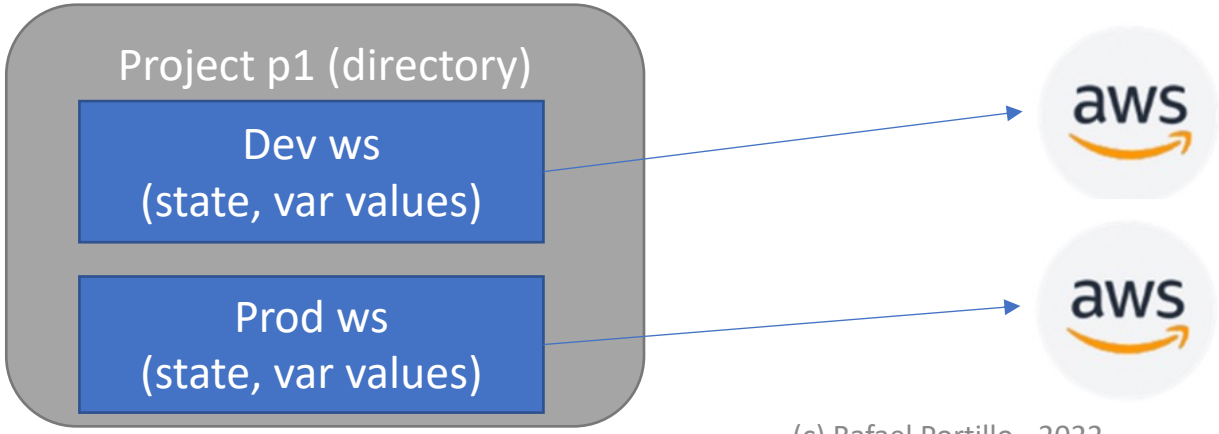
- file provisioner (Docs)
  - Copies files or directories from the machine running Terraform to the newly created resource
- remote-exec provisioner (Docs)
  - Invokes a script on a remote resource after it is created
- These two provisioners often need a means to connect to the remote server (VM, etc.) – this is configured under the "connection" block
  - Example shows user and password but SSH private key also supported.

```
# 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}"
  }
}
```

Terraform (CLI) Workspaces

Terraform CLI Workspaces - Intro

- Terraform Docs
- IMPORTANT: This section of the course discusses Terraform **CLI** Workspaces, not to be confused with Terraform Cloud Workspaces
- Purpose of Workspaces
  - Provide different “environments” under a common directory
    - Each workspace has own state, variables, etc..



Terraform CLI Workspaces – Additional info

- HashiCorp documentation discusses at length when NOT to use multiple Workspaces and alternatives
  - Main Problem: same backend
    - "Workspaces alone are not a suitable tool for system decomposition because each subsystem should have its own separate configuration and backend."
    - "CLI workspaces within a working directory use the same backend, so they are not a suitable isolation mechanism for this scenario."
- Alternatives – among others:
  - Different configuration (directories) with re-usable modules (e.g. in repo, such as github)

Demo Lab (AWS)

- Lab 11 – workspaces
- Suggested Activities
  - Explore workspace CLI commands
  - Try terraform plan in each of the workspaces (including default)
  - Explore .terraform.tfstate.d
    - Where is the state for the default workspace?
  - Create a new workspace (test) with t3.micro instance and 3 instance types
  - Consider making the number of Azs also depending on the workspace
    - E.g. 1 for dev, 2 for test, 2 for prod
  - Note: name_suffix includes the workspace name

Name	Instance ID
vm-proj99-dev-11ws-default-0	i-0f812eaf835aea26b
vm-proj99-dev-11ws-dev-0	i-042fb27b1ab23377c
vm-proj99-dev-11ws-prod-0	i-090430105a0c25c7b
vm-proj99-dev-11ws-prod-1	i-01654eb58b6708a9c

```
variable "instance_type" {
  type = map
  default = {
    default = "t2.micro"
    dev     = "t2.micro"
    prod    = "t3.micro"
  }
}

variable "num_instances" {
  type = map

  default = {
    default = 1
    dev     = 1
    prod    = 2
  }
}
```

```
output workspace_name {
  description = "Active workspace"
  value = terraform.workspace
}
output num_instances_ws {
  description = "Number of instances for this workspace"
  value = lookup(var.num_instances, terraform.workspace)
}

output instance_type_ws {
  description = "Instance type for this workspace"
  value = lookup(var.instance_type, terraform.workspace)
}
```

```
resource "aws_instance" "server" {
  count         = lookup(var.num_instances, terraform.workspace)
  instance_type = lookup(var.instance_type, terraform.workspace)
  ami           = data.aws_ami.amazon_linux2_kernel_5.id

  subnet_id = data.aws_subnets.def_vpc_subnets.ids[count.index % var.num_azs]

  tags = {
    Name = "vm-${local.name_suffix}-${count.index}"
  }
}
```

Other Topics

Sensitive Values in Terraform

- ”sensitive” flag – helps us “hide” sensitive values form TF output
- The value will be “hidden” in output of commands like plan and command
- The “sensitive” flag can be used in :
  - Variables
  - Outputs
  - Providers
  - Modules
  - Functions
- Sensitive values are NOT hidden in the state file.
- Related: *nonsensitive* function
  - Takes a sensitive value as parameter and returns the same value with the sensitive flag removed – exposing its value

```
variable "web_secret" {
    description = "Required for login"
    value = module.my_server.web_secret
    sensitive = true
}
```

```
output "server_pw" {
    description = "Password for DB"
    value = aws_db_instance.password
    sensitive = true
}
```