**Terraform Provider Versioning - Lab Guide**

**Objective**

This lab guide will help you understand and experiment with Terraform provider versioning. You will learn how to check and control provider versions, upgrade/downgrade versions, and resolve potential conflicts.

**Prerequisites**

* **Terraform Installed**: Ensure Terraform is installed on your system. You can check by running:
* terraform version
* **AWS CLI Configured**: You should have AWS credentials configured in ~/.aws/credentials or use environment variables.
* **An AWS Account**: Ensure you have an AWS account with necessary permissions.
* **Basic Terraform Knowledge**: Familiarity with Terraform configuration files and commands.

**Step 1: Create Terraform Configuration**

1. **Create a working directory**
2. mkdir terraform-provider-lab && cd terraform-provider-lab
3. **Create a res.tf file with the following content:**
4. provider "aws" {
5. region = "us-east-2"
6. version = ">=2.10,<=2.30"
7. }
8. resource "aws\_instance" "myawsserver" {
9. ami = "ami-064ff912f78e3e561"
10. instance\_type = "t2.micro"
11. tags = {
12. Name = "Techlanders-aws-ec2-instance"
13. }
14. }

**Step 2: Check the Current Version**

1. Run the following command to check the locked provider version:
2. cat .terraform.lock.hcl

If the file does not exist, run terraform init first.

1. The output will show the specific AWS provider version currently in use.

**Step 3: Initialize Terraform**

1. Initialize Terraform to download the required provider version:
2. terraform init
3. Verify that Terraform has downloaded the correct provider version by checking the output of:
4. terraform providers

**Step 4: Upgrade/Downgrade the Provider Version**

1. **Delete the lock file (if needed)**
2. rm -f .terraform.lock.hcl
3. **Try upgrading the provider**
4. terraform init -upgrade

This will upgrade the AWS provider to the latest version that matches the constraints specified in res.tf.

1. **Modify the provider version constraint**
   * Change the version constraint in res.tf to test different behaviors. Examples:
   * version = "~> 5.x" # Allows automatic updates within 5.x

or

version = "<=5.92.0" # Restricts to 5.92.0 or below

* + Run terraform init -upgrade again to see how the provider updates.

**Step 5: Experimenting with Provider Constraints**

1. Set a strict version constraint and initialize:
2. version = "=5.91.0"
3. terraform init

Terraform will install exactly **5.91.0**.

1. Change to a **more flexible constraint**:
2. version = "~> 5.x"
3. terraform init -upgrade

Terraform will upgrade to the latest **5.x** version available.

1. **Test what happens when you remove the version constraint entirely**:
2. provider "aws" {
3. region = "us-east-2"
4. }
5. terraform init -upgrade

Terraform will install the latest available version.

**Step 6: Cleanup**

To clean up resources created during this lab:

1. **Destroy the infrastructure**:
2. terraform destroy -auto-approve
3. **Remove Terraform files**:
4. rm -rf .terraform\* terraform.tfstate\* .terraform.lock.hcl

**Conclusion**

In this lab, you:

* Checked the current Terraform provider version.
* Used constraints to control provider versions.
* Upgraded and downgraded provider versions.
* Learned how terraform init -upgrade works.

This knowledge is essential when working in production environments to ensure compatibility and avoid breaking changes!

**Terraform Variables - Lab Guide**

**Objective**

This lab guide will help you understand and experiment with Terraform variables, including different types, variable assignments, and modifying values dynamically.

**Prerequisites**

* **Terraform Installed**: Ensure Terraform is installed. Check with:
* terraform version
* **AWS CLI Configured**: Ensure AWS credentials are configured.
* **An AWS Account**: Necessary permissions to launch EC2 and create security groups.
* **Basic Terraform Knowledge**: Familiarity with Terraform configuration files.

**Step 1: Create Terraform Configuration**

1. **Create a working directory**:
2. mkdir terraform-variables-lab && cd terraform-variables-lab
3. **Create a variables.tf file to define input variables**:
4. variable "rk-cidr" {
5. type = string
6. default = "116.50.30.70/32" # Restrict access to a specific IP
7. }
8. **Create a res.tf file with the following content**:
9. provider "aws" {
10. region = "us-east-1"
11. }
12. resource "aws\_security\_group" "rksg" {
13. name = "raman-variables"
14. vpc\_id = "vpc-01948378f1e13345b"
15. ingress {
16. from\_port = 443
17. to\_port = 443
18. protocol = "tcp"
19. cidr\_blocks = [var.rk-cidr]
20. }
21. ingress {
22. from\_port = 80
23. to\_port = 80
24. protocol = "tcp"
25. cidr\_blocks = [var.rk-cidr]
26. }
27. ingress {
28. from\_port = 53
29. to\_port = 53
30. protocol = "tcp"
31. cidr\_blocks = [var.rk-cidr]
32. }
33. ingress {
34. from\_port = 3389
35. to\_port = 3389
36. protocol = "tcp"
37. cidr\_blocks = [var.rk-cidr]
38. }
39. }

**Step 2: Initialize and Apply Terraform**

1. Initialize Terraform:
2. terraform init
3. Apply the configuration:
4. terraform apply -auto-approve

Terraform will create the security group with rules based on rk-cidr.

**Step 3: Modify Variable Values**

**1. Change the Default Value in variables.tf**

Modify the default value of rk-cidr in variables.tf:

variable "rk-cidr" {

type = string

default = "0.0.0.0/0" # Allow access from anywhere

}

Run Terraform again:

terraform apply -auto-approve

This will update the security group rules.

**2. Assign Values via CLI**

You can override the default value using the -var flag:

terraform apply -var="rk-cidr=192.168.1.1/32" -auto-approve

This allows customization without modifying variables.tf.

**3. Using a .tfvars File**

Create a custom.tfvars file:

rk-cidr = "203.0.113.10/32"

Apply Terraform using this file:

terraform apply -var-file="custom.tfvars" -auto-approve

This is useful for managing multiple environments (e.g., dev, staging, production).

**4. Using Environment Variables**

Set an environment variable to assign a value:

export TF\_VAR\_rk-cidr="192.168.10.0/24"

Run Terraform:

terraform apply -auto-approve

**Step 4: Working with Different Variable Types**

Terraform supports multiple variable types:

**String Variable**

variable "region" {

type = string

default = "us-east-1"

}

**Number Variable**

variable "instance\_count" {

type = number

default = 2

}

**Boolean Variable**

variable "enable\_logs" {

type = bool

default = true

}

**List Variable**

variable "allowed\_ports" {

type = list(number)

default = [80, 443, 22]

}

**Map Variable**

variable "instance\_types" {

type = map(string)

default = {

dev = "t2.micro"

prod = "t3.medium"

}

}

To reference in resources:

instance\_type = var.instance\_types["dev"]

**Step 5: Cleanup**

To remove the created resources:

terraform destroy -auto-approve

Delete Terraform files:

rm -rf .terraform\* terraform.tfstate\* .terraform.lock.hcl

**Conclusion**

In this lab, you:

* Defined Terraform variables and assigned values.
* Modified variables dynamically using CLI, .tfvars, and environment variables.
* Used different variable types like string, number, boolean, list, and map.

Understanding Terraform variables is crucial for writing reusable and scalable infrastructure code. 🚀

**Terraform Variable Assignment - Lab Guide**

**Objective**

This lab guide explores different approaches for assigning variables in Terraform, including default values, .tfvars files, and CLI overrides.

**Prerequisites**

* **Terraform Installed**: Ensure Terraform is installed:
* terraform version
* **AWS CLI Configured**
* **An AWS Account**
* **Basic Terraform Knowledge**

**Step 1: Create Terraform Configuration**

**1. Define variable.tf File**

Create a variable.tf file to define variables:

variable "typ" {

default = "t2.micro" # Default instance type

}

variable "rk-cidr" {

type = string

default = "116.50.30.70/32" # Default CIDR block

}

**2. Create res.tf with Resources**

provider "aws" {

region = "us-east-1"

}

resource "aws\_instance" "myawsserver" {

ami = "ami-04aa00acb1165b32a"

instance\_type = var.typ

vpc\_security\_group\_ids = [aws\_security\_group.rksg.id]

tags = {

Name = "raman-aws-instance"

}

}

resource "aws\_security\_group" "rksg" {

name = "raman-variables"

vpc\_id = "vpc-01948378f1e13345b"

ingress {

from\_port = 443

to\_port = 443

protocol = "tcp"

cidr\_blocks = [var.rk-cidr]

}

ingress {

from\_port = 80

to\_port = 80

protocol = "tcp"

cidr\_blocks = [var.rk-cidr]

}

ingress {

from\_port = 53

to\_port = 53

protocol = "tcp"

cidr\_blocks = [var.rk-cidr]

}

ingress {

from\_port = 3389

to\_port = 3389

protocol = "tcp"

cidr\_blocks = [var.rk-cidr]

}

}

**Step 2: Different Ways to Assign Variables**

**1. Using Default Values in variable.tf**

Terraform uses default values when no other input is provided:

terraform apply -auto-approve

**2. Overriding Variables Using .tfvars Files**

Create a custom ramanprod.tfvars file:

typ="t2.large"

rk-cidr="0.0.0.0/0"

Apply using:

terraform apply -var-file="ramanprod.tfvars" -auto-approve

**3. Using terraform.tfvars File**

Terraform automatically reads terraform.tfvars if it exists in the working directory.

Example terraform.tfvars:

typ="t2.nano"

Terraform will use values from terraform.tfvars unless overridden by another method.

**4. Using CLI Overrides**

Specify values directly in the command line:

terraform apply -var="typ=t2.small" -auto-approve

CLI values override .tfvars and default values.

**5. Using Environment Variables**

Export an environment variable:

export TF\_VAR\_typ="t2.medium"

terraform apply -auto-approve

Environment variables take precedence over defaults but are overridden by CLI values.

**Step 3: Verify Variable Assignments**

Check assigned values:

terraform plan

The output will display the instance type and security group CIDR block in use.

**Step 4: Cleanup**

Destroy resources:

terraform destroy -auto-approve

Delete files:

rm -rf .terraform\* terraform.tfstate\* .terraform.lock.hcl

**Conclusion**

In this lab, you learned:

* How Terraform assigns default values to variables.
* How to override defaults using .tfvars, terraform.tfvars, CLI, and environment variables.
* The priority order of variable assignments in Terraform.

This knowledge helps in managing configurations efficiently across multiple environments. 🚀

**Terraform Count Parameter and Indexes - Lab Guide**

**Objective**

This lab demonstrates the use of the count parameter and index-based variable assignments in Terraform to create multiple resources dynamically.

**Prerequisites**

* **Terraform Installed**: Verify with:
* terraform version
* **AWS CLI Configured**
* **An AWS Account**
* **Basic Terraform Knowledge**

**Step 1: Define Variables**

**1. Create variable.tf File**

Define variables for instance types, security group rules, and server names:

variable "typ" {

type = list(string)

default = ["t2.micro", "t2.medium", "t2.large"]

}

variable "rk-cidr" {

type = string

default = "116.50.30.70/32"

}

variable "serv-names" {

type = list(string)

default = ["dev", "stage", "prod"]

}

**Step 2: Define Resources with Count**

**1. Create res.tf File**

Define AWS provider and dynamically create instances using count:

provider "aws" {

region = "us-east-1"

}

resource "aws\_instance" "myawsserver" {

ami = "ami-04aa00acb1165b32a"

instance\_type = var.typ[count.index]

count = 3

vpc\_security\_group\_ids = [aws\_security\_group.rksg.id]

tags = {

Name = "raman-${var.serv-names[count.index]}-server"

}

}

resource "aws\_security\_group" "rksg" {

name = "raman-variables"

vpc\_id = "vpc-01948378f1e13345b"

ingress {

from\_port = 443

to\_port = 443

protocol = "tcp"

cidr\_blocks = [var.rk-cidr]

}

ingress {

from\_port = 80

to\_port = 80

protocol = "tcp"

cidr\_blocks = [var.rk-cidr]

}

ingress {

from\_port = 53

to\_port = 53

protocol = "tcp"

cidr\_blocks = [var.rk-cidr]

}

ingress {

from\_port = 3389

to\_port = 3389

protocol = "tcp"

cidr\_blocks = [var.rk-cidr]

}

}

**Step 3: Initialize and Apply Terraform**

**1. Initialize Terraform**

terraform init

**2. Plan Deployment**

terraform plan

**3. Apply Terraform Configuration**

terraform apply -auto-approve

This will create **3 EC2 instances** with instance types and names based on the indexed variables:

* t2.micro → raman-dev-server
* t2.medium → raman-stage-server
* t2.large → raman-prod-server

**Step 4: Verify Deployment**

Check the AWS Console or use:

aws ec2 describe-instances --query 'Reservations[\*].Instances[\*].Tags[\*]'

**Step 5: Cleanup Resources**

Destroy all resources:

terraform destroy -auto-approve

Delete generated files:

rm -rf .terraform\* terraform.tfstate\* .terraform.lock.hcl

**Conclusion**

In this lab, you learned:

* How to use the count parameter to create multiple resources dynamically.
* How to reference list variables using count.index.
* How to apply variable-based configurations for dynamic deployments.

This method is useful for scaling environments efficiently using Terraform! 🚀

**Terraform Dynamic Resource Creation with Maps and Conditionals - Lab Guide**

**Objective**

This lab demonstrates how to use for\_each, dynamic blocks, maps, lists, and conditionals in Terraform to manage AWS infrastructure efficiently.

**Prerequisites**

* **Terraform Installed**: Verify with:
* terraform version
* **AWS CLI Configured**
* **An AWS Account**
* **Basic Terraform Knowledge**

**Step 1: Define Variables**

**1. Create res2.tf File**

Define variables for environment selection, instance types, security group rules, and conditional logic:

provider "aws" {

region = "us-east-1"

}

variable "env" {

description = "Deployment environment"

type = string

default = "dev"

}

variable "instances" {

type = map(string)

default = {

"web" = "t2.micro"

"db" = "t3.medium"

"worker" = "t2.small"

}

}

variable "cidr\_blocks" {

type = list(string)

default = ["0.0.0.0/0"]

}

variable "sg\_ports" {

type = list(number)

description = "List of ingress ports"

default = [8200, 8201, 8300, 9200, 9500]

}

variable "use\_premium\_storage" {

description = "Enable premium storage for prod instances"

type = bool

default = false

}

**Step 2: Define Resources**

**1. Define AWS Security Group with Dynamic Ingress Rules**

resource "aws\_security\_group" "rksg" {

name = "raman-secgroup"

vpc\_id = "vpc-01948378f1e13345b"

dynamic "ingress" {

for\_each = var.sg\_ports

content {

from\_port = ingress.value

to\_port = ingress.value

protocol = "tcp"

cidr\_blocks = var.cidr\_blocks

}

}

}

**2. Create EC2 Instances Dynamically Using for\_each**

resource "aws\_instance" "myawsserver" {

for\_each = var.instances

ami = "ami-04aa00acb1165b32a"

instance\_type = var.env == "prod" ? "t3.large" : each.value

root\_block\_device {

volume\_type = var.use\_premium\_storage ? "io1" : "gp2"

}

vpc\_security\_group\_ids = [aws\_security\_group.rksg.id]

tags = {

Name = "raman-${each.key}-server"

Env = var.env

}

}

**Step 3: Initialize and Apply Terraform**

**1. Initialize Terraform**

terraform init

**2. Plan Deployment**

terraform plan

**3. Apply Terraform Configuration**

terraform apply -auto-approve

This will create:

* **Security Group** with **dynamic ingress rules** based on sg\_ports
* **EC2 instances** dynamically named raman-web-server, raman-db-server, and raman-worker-server
* **Conditional instance types**:
  + Default: Uses values from instances
  + If env is prod, all instances are t3.large
  + Uses io1 storage if use\_premium\_storage is true

**Step 4: Verify Deployment**

Check the AWS Console or use:

aws ec2 describe-instances --query 'Reservations[\*].Instances[\*].Tags[\*]'

**Step 5: Cleanup Resources**

Destroy all resources:

terraform destroy -auto-approve

Delete generated files:

rm -rf .terraform\* terraform.tfstate\* .terraform.lock.hcl

**Conclusion**

In this lab, you learned:

* How to use for\_each to dynamically create multiple EC2 instances.
* How to use **dynamic blocks** to generate ingress rules for security groups.
* How to use **conditional expressions** for environment-based instance types.
* How to control **storage types** using a boolean variable.

This approach makes Terraform configurations highly **scalable and flexible**! 🚀

**Terraform Lab Guide: Working with Data Sources in AWS**

**Overview**

In this lab, you will learn how to use Terraform data sources to fetch information about existing AWS resources dynamically. This will help in creating new resources based on existing configurations.

**Prerequisites**

* AWS account
* Terraform installed
* AWS CLI configured with credentials

**Step 1: Define the AWS Provider**

Create a Terraform configuration file (e.g., ec2.tf) and specify the AWS provider settings.

provider "aws" {

region = "us-east-1"

access\_key = "<YOUR\_ACCESS\_KEY>"

secret\_key = "<YOUR\_SECRET\_KEY>"

}

**Note:** Replace <YOUR\_ACCESS\_KEY> and <YOUR\_SECRET\_KEY> with your actual AWS credentials.

**Step 2: Define a Data Source for an Existing EC2 Instance**

If you already have an instance running and want to fetch its properties, use the aws\_instance data source:

data "aws\_instance" "foo" {

instance\_id = "i-050a18394790a1cc8"

}

This fetches details about the EC2 instance with ID i-050a18394790a1cc8.

**Step 3: Define a Data Source for the Latest Amazon Linux 2 AMI**

You can dynamically retrieve the latest Amazon Linux 2 AMI using the aws\_ami data source:

data "aws\_ami" "app\_ami" {

most\_recent = true

owners = ["amazon"]

filter {

name = "name"

values = ["amzn2-ami-hvm\*"]

}

}

This fetches the most recent Amazon Linux 2 AMI.

**Step 4: Create a New EC2 Instance Using Retrieved Data**

Now, create an EC2 instance using the data fetched from the above sources:

resource "aws\_instance" "ec2" {

ami = data.aws\_ami.app\_ami.id

instance\_type = data.aws\_instance.foo.instance\_type

availability\_zone = data.aws\_instance.foo.availability\_zone

tags = {

Name = "raman-ec2"

}

}

This will create a new instance using:

* The latest Amazon Linux 2 AMI
* The instance type and availability zone of the existing EC2 instance

**Step 5: Using Data Sources in Another Region**

To use data sources in another AWS region (e.g., ap-south-1), modify the provider:

provider "aws" {

region = "ap-south-1"

access\_key = "<YOUR\_ACCESS\_KEY>"

secret\_key = "<YOUR\_SECRET\_KEY>"

}

Then, fetch the latest Amazon Linux 2 AMI and launch an EC2 instance in ap-south-1:

data "aws\_ami" "app\_ami" {

most\_recent = true

owners = ["amazon"]

filter {

name = "name"

values = ["amzn2-ami-hvm\*"]

}

}

resource "aws\_instance" "ec2" {

ami = data.aws\_ami.app\_ami.id

instance\_type = "t2.micro"

tags = {

Name = "raman-ec2"

}

}

**Step 6: Deploy the Terraform Configuration**

Run the following commands to initialize Terraform, plan the deployment, and apply the changes:

terraform init

terraform plan

terraform apply -auto-approve

After successful deployment, Terraform will output details about the created EC2 instance.

**Step 7: Cleanup**

To delete the created resources, run:

terraform destroy -auto-approve

This will remove all resources created by Terraform.

**Summary**

* Used Terraform data sources to retrieve existing EC2 instance details
* Dynamically retrieved the latest Amazon Linux 2 AMI
* Created a new EC2 instance using retrieved data
* Deployed and cleaned up resources using Terraform

This approach makes Terraform configurations more dynamic and reusable! 🚀