**Terraform Lab Guide: Resource Meta-Argument - lifecycle**

**Objective**

In this lab, you will learn how to use the lifecycle meta-argument in Terraform to:

1. Control resource destruction and recreation behavior.
2. Prevent accidental deletion of critical resources.
3. Ignore specific changes to avoid unintended updates.

**Lab Prerequisites**

✅ Terraform installed (v1.3+ recommended)  
✅ AWS CLI configured with appropriate credentials  
✅ AWS Account with permissions to create EC2 instances and S3 buckets  
✅ Basic knowledge of Terraform syntax

**Lab Structure**

You will perform the following exercises:

* **Exercise 1:** Implement create\_before\_destroy
* **Exercise 2:** Implement prevent\_destroy
* **Exercise 3:** Implement ignore\_changes

**Exercise 1: Ensuring Zero Downtime Using create\_before\_destroy**

**Use Case**

You want to replace an EC2 instance without downtime. Terraform should create the new instance before destroying the old one.

**Steps**

1. **Create the Terraform Configuration** Create a directory for this exercise:

bash

CopyEdit

mkdir terraform-lifecycle-lab

cd terraform-lifecycle-lab

1. **Create a file named ec2-instance.tf**

hcl

CopyEdit

provider "aws" {

region = "us-east-1"

}

resource "aws\_instance" "example" {

ami = "ami-0c55b159cbfafe1f0" # Update this with a valid AMI ID in your region

instance\_type = "t2.micro"

lifecycle {

create\_before\_destroy = true

}

tags = {

Name = "example-instance"

}

}

1. **Initialize and Apply**

bash

CopyEdit

terraform init

terraform apply -auto-approve

1. **Test Behavior**
   * Modify the AMI ID in ec2-instance.tf to a different valid AMI.
   * Run terraform apply again.
   * Observe: A new instance is created **before** destroying the old one.

**Exercise 2: Prevent Accidental Deletion Using prevent\_destroy**

**Use Case**

You want to prevent accidental deletion of an S3 bucket containing critical data.

**Steps**

1. **Create a file named s3-bucket-prevent-destroy.tf**

hcl

CopyEdit

resource "aws\_s3\_bucket" "critical" {

bucket = "critical-bucket-${random\_id.bucket\_suffix.hex}"

lifecycle {

prevent\_destroy = true

}

tags = {

Purpose = "Critical data storage"

}

}

resource "random\_id" "bucket\_suffix" {

byte\_length = 4

}

1. **Initialize and Apply**

bash

CopyEdit

terraform init

terraform apply -auto-approve

1. **Test Behavior**
   * Try to delete the bucket:

bash

CopyEdit

terraform destroy

* + **Expected Result:** Terraform will fail with an error:

javascript

CopyEdit

Error: Instance cannot be destroyed

* + To allow destruction, you must **manually remove** prevent\_destroy or set it to false.

**Exercise 3: Ignoring Specific Changes Using ignore\_changes**

**Use Case**

You want to ignore any tag changes for Owner on an S3 bucket to avoid unnecessary updates.

**Steps**

1. **Create a file named s3-bucket-ignore-changes.tf**

hcl

CopyEdit

resource "aws\_s3\_bucket" "example" {

bucket = "mutable-bucket-${random\_id.bucket\_suffix.hex}"

tags = {

Environment = "Dev"

Owner = "DevOps"

}

lifecycle {

ignore\_changes = [

tags["Owner"]

]

}

}

resource "random\_id" "bucket\_suffix" {

byte\_length = 4

}

1. **Initialize and Apply**

bash

CopyEdit

terraform init

terraform apply -auto-approve

1. **Test Behavior**
   * Modify the Owner tag in s3-bucket-ignore-changes.tf file to something else (e.g., "CloudOps").
   * Run terraform apply.
   * **Expected Result:** Terraform will **not** attempt to update the Owner tag because of ignore\_changes.

**Validation & Cleanup**

* You can run terraform state show <resource> to verify the state behavior.
* Clean up all resources after the lab:
  + For prevent\_destroy, first remove the prevent\_destroy block, re-apply, and then destroy.
  + For others:

bash

CopyEdit

terraform destroy

**✅ Key Takeaways**

| **Meta-Argument** | **Purpose** |
| --- | --- |
| create\_before\_destroy | Avoid downtime, ensures new resource is created first |
| prevent\_destroy | Protects critical resources from accidental deletion |
| ignore\_changes | Prevents unwanted updates for specified attributes |

**Terraform Lab Guide: Working with Functions (lookup, element, local, output formatting)**

**Objective**

This lab will teach you how to use the following Terraform concepts: ✅ Using lookup() function to fetch values dynamically from maps  
✅ Using element() function to fetch values from lists based on index  
✅ Using local values and timestamp() for metadata  
✅ Using structured output to extract resource details  
✅ Real-time demo with EC2 instance provisioning

**Lab Pre-requisites**

* AWS CLI configured with necessary permissions
* Terraform installed (v1.3+ recommended)
* AWS Account with EC2 provisioning rights
* Basic knowledge of Terraform variables, resources, and outputs

**Lab Topology & Goal**

You will provision **2 EC2 instances** in the us-east-1 region with dynamically assigned names and AMI IDs based on variables & functions.  
After deployment, Terraform will display a **Consolidated Output** showing:

* Instance IDs
* Public IPs
* Private IPs
* AMI IDs
* Instance Names (Tags)
* Timestamp of deployment

**Lab Structure**

**1. Project Setup**

Create a directory:

bash

CopyEdit

mkdir ~/terraform-functions-lab

cd ~/terraform-functions-lab

Create a file named func.tf and copy the following content:

hcl

CopyEdit

provider "aws" {

region = var.region

}

locals {

time = formatdate("DD MMM YYYY hh:mm ZZZ", timestamp())

}

variable "region" {

default = "us-east-1"

}

variable "tags" {

type = list(string)

default = ["raman-firstec2", "raman-secondec2"]

}

variable "ami" {

type = map(string)

default = {

"us-east-1" = "ami-08b5b3a93ed654d19"

"us-west-2" = "ami-0b6d6dacf350ebc82"

"ap-south-1" = "ami-05c179eced2eb9b5b"

}

}

resource "aws\_instance" "app-dev" {

ami = lookup(var.ami, var.region)

instance\_type = "t2.micro"

count = 2

tags = {

Name = element(var.tags, count.index)

}

}

output "instance\_details" {

description = "Consolidated details of all instances"

value = {

ids = aws\_instance.app-dev[\*].id

public\_ips = aws\_instance.app-dev[\*].public\_ip

private\_ips = aws\_instance.app-dev[\*].private\_ip

amis = aws\_instance.app-dev[\*].ami

instance\_names = aws\_instance.app-dev[\*].tags.Name

timestamp = local.time

}

}

**2. Key Terraform Function Explanation**

| **Function** | **Purpose** |
| --- | --- |
| lookup() | Dynamically fetches AMI ID from the map variable based on selected region. |
| element() | Fetches a specific element from a list using count.index. |
| local value | Stores formatted timestamp value using formatdate() and timestamp(). |
| output | Displays multiple instance details after apply. |

**3. Initialize Terraform**

bash

CopyEdit

terraform init

**4. Validate the Configuration**

bash

CopyEdit

terraform plan

You should see that Terraform is planning to create **2 EC2 instances** with dynamically assigned names.

**5. Deploy the Infrastructure**

bash

CopyEdit

terraform apply -auto-approve

✅ **Expected Result**  
2 EC2 instances created with the following tag names:

* raman-firstec2
* raman-secondec2

**6. Verify Outputs**

Run:

bash

CopyEdit

terraform output

Sample Output:

json

CopyEdit

instance\_details = {

"ids" = [

"i-0a1b2c3d4e5f6g7h8",

"i-0h7g6f5e4d3c2b1a0"

]

"public\_ips" = [

"34.123.45.67",

"34.87.65.43"

]

"private\_ips" = [

"172.31.16.21",

"172.31.18.34"

]

"amis" = [

"ami-08b5b3a93ed654d19",

"ami-08b5b3a93ed654d19"

]

"instance\_names" = [

"raman-firstec2",

"raman-secondec2"

]

"timestamp" = "27 Mar 2025 02:31 +0000"

}

You can also export the output in JSON format:

bash

CopyEdit

terraform output -json > export.json

cat export.json

**7. Exploring Terraform Console (Optional)**

You can explore the functions interactively:

bash

CopyEdit

terraform console

> lookup({us-east-1="ami1",us-west-2="ami2"},"us-west-2","NA")

"ami2"

> element(["first","second","third"],1)

"second"

This will give learners practical exposure to these functions.

**8. Clean Up**

Destroy all resources:

bash

CopyEdit

terraform destroy -auto-approve

If destroy fails (due to dependencies or AWS termination protection), you can inspect the state:

bash

CopyEdit

terraform state list

**✅ Key Takeaways**

| **Terraform Concept** | **Description** |
| --- | --- |
| **lookup()** | Safe fetch from map variable with optional default value |
| **element()** | Fetch from a list based on index (used with count.index) |
| **local** | Used to store derived values (e.g., formatted timestamp) |
| **output block** | Consolidates resource attributes for visibility & automation |

**⭐ Lab Extension (Optional for Advanced Learners)**

1. Add a **third instance** and dynamically adjust the tags list.
2. Experiment with default value in lookup() function.
3. Add depends\_on and lifecycle meta-arguments.
4. Pipe output data into a Bash script or Ansible inventory.

**🚀 Terraform Lab Guide: Remote Backend and State Locking**

**Objective**

By the end of this lab, learners will: ✅ Understand how Terraform locks state files  
✅ Configure **S3** as a **Remote Backend**  
✅ Enable **Versioning** of the state file  
✅ Implement **State Locking** with **DynamoDB**  
✅ Observe state lock behavior using multiple sessions

**📝 Lab Pre-Requisites**

* AWS Account with admin privileges
* AWS CLI configured
* Terraform installed
* Basic knowledge of Terraform workflows

**🎯 Lab Workflow**

**1. Create an S3 Bucket for Remote Backend**

Create an S3 bucket:

bash

CopyEdit

aws s3api create-bucket --bucket raman-remote-backend-bucket --region us-east-1

✅ **Optional but Recommended:** Enable versioning on the bucket to protect against accidental deletions or overwrites:

bash

CopyEdit

aws s3api put-bucket-versioning --bucket raman-remote-backend-bucket --versioning-configuration Status=Enabled

**2. Create DynamoDB Table for State Locking**

State locking prevents simultaneous modifications to the same state file.

Create the table:

bash

CopyEdit

aws dynamodb create-table \

--table-name raman-tf-lock-table \

--attribute-definitions AttributeName=LockID,AttributeType=S \

--key-schema AttributeName=LockID,KeyType=HASH \

--provisioned-throughput ReadCapacityUnits=1,WriteCapacityUnits=1

**3. Configure Backend in Terraform**

In your working directory, create a file named backend.tf:

hcl

CopyEdit

terraform {

backend "s3" {

bucket = "raman-remote-backend-bucket"

key = "terraform/state"

region = "us-east-1"

dynamodb\_table = "raman-tf-lock-table"

}

}

**4. Initialize Terraform with Remote Backend**

Run:

bash

CopyEdit

terraform init

You should see:

nginx

CopyEdit

Terraform has been successfully initialized!

**Note:** This will create a terraform.tfstate file in S3 and DynamoDB will be used to lock/unlock the state.

**5. Apply a Sample Resource**

Create a simple resource file, main.tf:

hcl

CopyEdit

provider "aws" {

region = "us-east-1"

}

resource "aws\_s3\_bucket" "demo-bucket" {

bucket = "raman-demo-bucket-${random\_id.suffix.hex}"

}

resource "random\_id" "suffix" {

byte\_length = 4

}

output "bucket\_name" {

value = aws\_s3\_bucket.demo-bucket.bucket

}

Apply:

bash

CopyEdit

terraform apply -auto-approve

✅ Verify the state file:

bash

CopyEdit

aws s3 ls s3://raman-remote-backend-bucket/terraform/

✅ Verify lock table:

bash

CopyEdit

aws dynamodb scan --table-name raman-tf-lock-table

**🔐 6. Testing State Locking in Action**

**Objective:** Demonstrate Terraform’s locking mechanism by running apply simultaneously in two sessions.

**Steps:**

**Session 1:**

Open one terminal and run:

bash

CopyEdit

terraform apply

During the **plan/apply**, leave it running and don’t complete the apply (wait on confirmation).

**Session 2:**

Open another terminal in the same directory and run:

bash

CopyEdit

terraform apply

**Expected Result in Session 2:** You will see an error like:

yaml

CopyEdit

Error: Error acquiring the state lock

Error message: ConditionalCheckFailedException: The conditional request failed

Lock Info:

ID: 1234abcd-5678-efgh-ijkl-9876543210

Path: terraform/state

Operation: OperationTypeApply

Who: root@ip-172-31-30-85

Version: x.y.z

Created: 2025-03-27 10:31:00.123 +0000 UTC

Info:

✅ **You have successfully tested remote state locking!**

**🔥 7. Clean-Up (Optional but Recommended)**

Delete resources:

bash

CopyEdit

terraform destroy -auto-approve

Delete DynamoDB table:

bash

CopyEdit

aws dynamodb delete-table --table-name raman-tf-lock-table

Delete S3 bucket and contents:

bash

CopyEdit

aws s3 rb s3://raman-remote-backend-bucket --force

**📌 Key Concepts Recap**

| **Concept** | **Purpose** |
| --- | --- |
| **Remote Backend (S3)** | Stores Terraform state remotely and securely |
| **State Locking (DynamoDB)** | Prevents concurrent changes to the same state file |
| **Versioning on S3** | Allows rollback or recovery of old state versions |
| **LockID in DynamoDB** | Unique ID to manage state lock records |

**⭐ Lab Extension (Optional)**

1. Configure **Terraform Cloud** as backend and compare with S3 + DynamoDB setup.
2. Explore **force-unlock**:

bash

CopyEdit

terraform force-unlock LOCK-ID

1. Create a Jenkins job to trigger Terraform apply with remote backend.

**🎯 Outcome**

By performing this lab, learners gain hands-on experience with one of the most important operational best practices in Terraform — **Remote State Management & State Locking** — which is mandatory in team environments.

**✅ Terraform Provisioners Lab Guide (Remote-Exec & Local-Exec)**

**Lab Objectives**

By the end of this lab, you will be able to:

* Understand the role and purpose of **Terraform Provisioners**
* Implement **Remote Exec Provisioner** to install and configure a webserver on EC2
* Implement **Local Exec Provisioner** to perform local machine operations after resource creation
* Recognize **Provisioner Best Practices** and when to use them properly
* Differentiate between **Infrastructure Creation** and **Configuration Management**

**🟣 Best Practices Background**

| **Provisioning Responsibility** | **Tool** |
| --- | --- |
| Infrastructure Creation | Terraform |
| Configuration Management | Ansible / Chef / Puppet / Salt |
| Small or One-Time Configurations | Terraform provisioners (Remote or Local Exec) |

⚡ **Important:**  
Terraform is not meant to be a full-fledged configuration management tool.  
Provisioners should only be used:

* For bootstrap tasks (installing agents)
* One-shot commands
* Avoid for complex configuration (use Ansible, etc.)

**Part 1: Remote Exec Provisioner Lab**

**✅ 1. Create Key Pair**

Create a key pair in AWS to SSH into EC2:

bash

CopyEdit

aws ec2 create-key-pair --key-name raman-virginia-key --query 'KeyMaterial' --output text > raman.pem

chmod 400 raman.pem

mv raman.pem /root/app1/nvirginia.pem

**✅ 2. Prepare remote.tf**

Create remote.tf with the following:

hcl

CopyEdit

provider "aws" {

region = "us-east-1"

}

resource "aws\_security\_group" "allow\_ssh" {

name = "allow\_ssh"

description = "Allow SSH and HTTP"

vpc\_id = "vpc-01948378f1e13345b" # Replace with your correct VPC ID

ingress {

description = "Allow SSH"

from\_port = 22

to\_port = 22

protocol = "tcp"

cidr\_blocks = ["0.0.0.0/0"]

}

ingress {

description = "Allow HTTP"

from\_port = 80

to\_port = 80

protocol = "tcp"

cidr\_blocks = ["0.0.0.0/0"]

}

egress {

description = "Allow all outbound"

from\_port = 0

to\_port = 65535

protocol = "tcp"

cidr\_blocks = ["0.0.0.0/0"]

}

}

resource "aws\_instance" "myec2" {

ami = "ami-04aa00acb1165b32a" # Amazon Linux 2

instance\_type = "t2.micro"

key\_name = "raman-virginia-key"

vpc\_security\_group\_ids = [aws\_security\_group.allow\_ssh.id]

tags = {

Name = "raman-web-server"

}

provisioner "remote-exec" {

inline = [

"sudo amazon-linux-extras install -y nginx1.12",

"sudo systemctl start nginx",

"sudo systemctl enable nginx"

]

connection {

type = "ssh"

user = "ec2-user"

private\_key = file("/root/app1/nvirginia.pem")

host = self.public\_ip

}

}

}

**✅ 3. Apply**

bash

CopyEdit

terraform init

terraform apply -auto-approve

**✅ 4. Test**

Get the EC2 public IP:

bash

CopyEdit

terraform output

Access:

bash

CopyEdit

curl http://<EC2\_PUBLIC\_IP>

You should see:

css

CopyEdit

HTML content of the NGINX default page

**Part 2: Local Exec Provisioner Lab**

**✅ 1. Prepare local.tf**

hcl

CopyEdit

provider "aws" {

region = "us-east-1"

}

resource "aws\_instance" "myec2" {

ami = "ami-04aa00acb1165b32a"

instance\_type = "t2.micro"

provisioner "local-exec" {

command = "echo ${self.private\_ip} >> private\_ips.txt"

}

}

**✅ 2. Apply**

bash

CopyEdit

terraform init

terraform apply -auto-approve

**✅ 3. Validate**

bash

CopyEdit

cat private\_ips.txt

✅ You will see:

cpp

CopyEdit

<private-ip-of-created-instance>

**Understanding What Happened**

| **Provisioner** | **Where it Runs** | **Purpose** |
| --- | --- | --- |
| **Remote Exec** | Inside the resource (EC2) | Configure EC2 (e.g., install NGINX) |
| **Local Exec** | On your local machine | Store instance information (e.g., IP addresses) |

**✅ Recommended Best Practices**

1. **Use provisioners only if there is no other choice.**
2. For large-scale or repeatable configurations, use **Ansible** or **other CM tools**.
3. Never manage full application configuration using Terraform provisioners.
4. Avoid sensitive data leakage through local-exec outputs.
5. Avoid depending on provisioners for resource dependency logic.

**✅ Bonus Exercise (Optional)**

You can extend this lab by:

1. Using file provisioner to copy scripts
2. Using Ansible instead of remote-exec for NGINX installation
3. Automating the provisioning using Jenkins + Terraform

**✅ Key Takeaways**

| **Concept** | **Explanation** |
| --- | --- |
| Remote-Exec | Executes on the EC2 after creation |
| Local-Exec | Executes on your laptop/local system |
| Configuration Management | Should be handled by Ansible, not Terraform provisioners |
| Best Practice | Use provisioners minimally, only for bootstrapping |