**Lab Guide: Implementing State Locking in Terraform with S3 and DynamoDB**

**Objective**

This guide provides detailed steps for implementing Terraform state file locking using Amazon S3 as the backend storage and DynamoDB for state locking. We will create an S3 bucket, a DynamoDB table for state locking, and configure Terraform to use them. Finally, we’ll verify the locking mechanism during a terraform apply.

**Prerequisites**

1. AWS account credentials with permissions to create S3 buckets, DynamoDB tables, and related resources.
2. Terraform installed on your system (version 0.12 or newer recommended).
3. An EC2 instance or local setup to execute Terraform commands.

**Steps to Create the Terraform Environment**

**1. Create S3 Bucket for State Storage**

1. Log into the AWS Management Console.
2. Navigate to **S3** service.
3. Click **Create bucket**.
   * Bucket name: raman-cisco-statefile (replace with your desired name if required).
   * Region: us-east-1.
   * Leave other options as default and click **Create bucket**.

**2. Create DynamoDB Table for Locking**

1. Navigate to the **DynamoDB** service in the AWS Console.
2. Click **Create table**.
   * Table name: tflock.
   * Partition key: LockID (Data type: String).
   * Leave other options as default and click **Create table**.

**3. Prepare the Terraform Backend Configuration File**

Create the backend.tf file with the following configuration:

terraform {

backend "s3" {

bucket = "raman-cisco-statefile"

key = "terraform/state"

region = "us-east-1"

dynamodb\_table = "tflock"

}

}

* **bucket**: The S3 bucket name for state storage.
* **key**: The path to store the state file.
* **region**: AWS region of the S3 bucket.
* **dynamodb\_table**: The DynamoDB table for state locking.

**4. Initialize the Terraform Backend**

Run the following command to initialize the backend:

terraform init

* Terraform will configure the S3 backend and link it to the DynamoDB table for locking.

**5. Create Sample Terraform Configuration**

1. Create a file named main.tf with a simple resource definition (e.g., an AWS S3 bucket):

provider "aws" {

region = "us-east-1"

}

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

bucket = "example-tflock-bucket"

acl = "private"

}

1. Validate the configuration:

terraform validate

**6. Test the Locking Mechanism**

1. Run terraform apply to provision resources:

terraform apply

* Terraform will check the DynamoDB table for an existing lock. If no lock exists, it will create one before applying changes.

1. Verify Lock Creation:
   * Go to the AWS Management Console and navigate to **DynamoDB** > **Tables** > tflock > \*\*Items`.
   * You should see an item created with details about the lock (e.g., LockID).
2. Simulate a Lock Conflict:
   * Open a second terminal and attempt to run terraform apply simultaneously.
   * The second attempt will block until the lock in DynamoDB is released.
3. Once the first terraform apply completes, the lock in DynamoDB will be removed automatically.

**7. Clean Up**

To avoid unnecessary costs, clean up the resources created:

1. Destroy Terraform-managed resources:

terraform destroy

1. Manually delete the S3 bucket and DynamoDB table if they are not managed by Terraform.

**Additional Notes**

* The locking mechanism ensures only one process can modify the Terraform state at a time, preventing race conditions and corruption.
* Always run terraform init after modifying the backend configuration.
* Ensure the S3 bucket and DynamoDB table have the correct permissions to avoid errors.

End of Lab Guide.

**Lab Guide: Implementing Lifecycle Meta-Arguments in Terraform**

**Objective**

This guide demonstrates the usage of Terraform lifecycle meta-arguments: create\_before\_destroy, prevent\_destroy, and ignore\_changes. These arguments help control the behavior of resources during creation, update, and deletion.

**Prerequisites**

1. AWS account credentials with permissions to create and manage EC2 instances and S3 buckets.
2. Terraform installed on your system (version 0.12 or newer recommended).
3. An EC2 instance or local setup to execute Terraform commands.

**Examples and Use Cases**

**1. create\_before\_destroy: Ensuring Zero Downtime**

**Use Case:**

Ensure zero downtime while replacing an EC2 instance.

**Configuration:**

1. Create a file ec2.tf with the following content:

provider "aws" {

region = "us-east-1"

}

resource "aws\_instance" "example" {

ami = "ami-123456"

instance\_type = "t2.micro"

lifecycle {

create\_before\_destroy = true

}

tags = {

Name = "example-instance"

}

}

1. Initialize and apply the configuration:

terraform init

terraform apply

**Test:**

* Modify the ami argument and run terraform apply. Notice that a new instance is created before the old one is destroyed.

**2. prevent\_destroy: Preventing Accidental Deletion**

**Use Case:**

Prevent accidental deletion of critical resources, such as an S3 bucket.

**Configuration:**

1. Create a file s3\_prevent\_destroy.tf with the following content:

provider "aws" {

region = "us-east-1"

}

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

bucket = "critical-bucket"

lifecycle {

prevent\_destroy = true

}

tags = {

Purpose = "Critical data storage"

}

}

1. Initialize and apply the configuration:

terraform init

terraform apply

**Test:**

* Run terraform destroy. Terraform will produce an error, indicating that the resource cannot be destroyed.
* To allow destruction, modify the lifecycle block to remove prevent\_destroy, apply the change, and then run terraform destroy again.

**3. ignore\_changes: Ignoring Specific Updates**

**Use Case:**

Avoid unintended updates to specific attributes, such as tags, during future Terraform runs.

**Configuration:**

1. Create a file s3\_ignore\_changes.tf with the following content:

provider "aws" {

region = "us-east-1"

}

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

bucket = "mutable-bucket"

tags = {

Environment = "Dev"

Owner = "DevOps"

}

lifecycle {

ignore\_changes = [

tags["Owner"]

]

}

}

1. Initialize and apply the configuration:

terraform init

terraform apply

**Test:**

* Update the tags["Owner"] value and run terraform apply. Terraform will not update the bucket resource for this change.

**Testing and Verification**

1. Use terraform plan to preview changes and ensure lifecycle arguments are applied as intended.
2. Monitor AWS resources in the Management Console to confirm behaviors.

**Clean-Up**

1. Run terraform destroy for all configurations (after removing prevent\_destroy where applicable).
2. Delete configuration files from your system to prevent inadvertent reuse.

**Summary**

* **create\_before\_destroy** ensures resources are replaced without downtime.
* **prevent\_destroy** safeguards critical resources from deletion.
* **ignore\_changes** prevents unnecessary updates to specified attributes.

End of Lab Guide.

**Lab Guide: Terraform Modules, Variables, and Outputs**

**Objective**

This guide demonstrates the use of modules, variables, and outputs in Terraform for reusable and flexible configurations. By following the lab instructions, you'll learn to:

1. Create reusable Terraform modules.
2. Use variables for parameterization.
3. Transfer values using output blocks between child and root modules.

**Prerequisites**

1. AWS account credentials with necessary permissions.
2. Terraform installed (version 0.12 or newer).
3. A setup environment (e.g., an EC2 instance or local machine).

**Part 1: Creating Reusable Modules**

**Step 1: Directory Structure**

1. Create the following directory structure:
2. modules/compute

projects/project-A

**Step 2: Create Source Module (ec2.tf)**

1. Navigate to modules/compute/ and create ec2.tf:

resource "aws\_instance" "s1" {

instance\_type = "t2.micro"

ami = "ami-064ff912f78e3e561"

}

**Step 3: Call Module from Child (myec2.tf)**

1. Navigate to projects/project-A and create myec2.tf:

module "ec2module" {

source = "../../modules/compute"

}

1. If errors occur, add a provider block in myec2.tf:

provider "aws" {

region = "us-east-1"

}

**Step 4: Run Terraform Commands**

1. Navigate to projects/project-A.
2. Initialize Terraform:

terraform init

1. Preview changes:

terraform plan

1. Apply changes:

terraform apply

**Part 2: Using Variables**

**Step 1: Update Source Module**

1. Modify modules/compute/ec2.tf to use variables:

resource "aws\_instance" "s1" {

instance\_type = var.instancetype

ami = "ami-064ff912f78e3e561"

}

1. Create a variables.tf file in the same directory:

variable "instancetype" {

default = "t2.micro"

}

**Step 2: Create Environment-Specific Configurations**

1. In projects/project-A, create myec2-dev.tf for the development environment:

module "ec2module-dev" {

source = "../../modules/compute"

instancetype = "t2.nano"

}

1. Create myec2-prod.tf for the production environment:

module "ec2module-prod" {

source = "../../modules/compute"

}

**Step 3: Run Terraform Commands for Each Environment**

1. Initialize and apply for the development environment:

terraform init

terraform apply -var-file=myec2-dev.tf

1. Repeat the process for production:

terraform apply -var-file=myec2-prod.tf

**Part 3: Using Output Blocks**

**Step 1: Modify Source Module to Include Outputs**

1. Update modules/compute/ec2.tf:

resource "aws\_instance" "myec2" {

ami = "ami-0e2c8caa4b6378d8c"

instance\_type = var.type

tags = {

Name = var.nam

}

}

output "type" {

value = var.type

description = "The instance type passed to the module"

}

1. Add variables in modules/compute/variables.tf:

variable "type" {}

variable "nam" {}

**Step 2: Access Outputs in Root Module**

1. In projects/project-A, create or update myec2.tf:

provider "aws" {

region = "us-east-1"

}

module "ec2module" {

source = "../../modules/compute"

type = var.type

nam = "projectB-server"

}

variable "type" {

description = "The instance type for the EC2 instance"

type = string

default = "t2.micro"

}

output "ec2module\_type" {

value = module.ec2module.type

}

**Step 3: Run Terraform Commands**

1. Initialize Terraform in projects/project-A:

terraform init

1. Apply the configuration and view the output:

terraform apply

**Clean-Up**

1. Destroy resources created during the lab:

terraform destroy

1. Delete configuration files to prevent reuse.

**Summary**

* **Modules** enable reusability and modularity in Terraform configurations.
* **Variables** provide flexibility and parameterization for dynamic configurations.
* **Outputs** allow value sharing between modules and root configurations.

End of Lab Guide.

**Lab Guide: Terraform Static Website Hosting with AWS S3**

**Objective**

This lab demonstrates the steps to create and deploy a static website hosted on Amazon S3 using Terraform, organized with reusable modules and configurations.

**Prerequisites**

1. AWS account credentials with appropriate permissions.
2. Terraform installed (version 0.12 or newer).
3. Basic understanding of Terraform and AWS S3.

**Directory Structure**

Organize your project directory as follows:

.

├── index.html

├── error.html

├── main.tf

├── modules

│ └── s3\_website

│ ├── main.tf

│ ├── variables.tf

│ └── outputs.tf

├── terraform.tfstate

└── terraform.tfstate.backup

**Step 1: HTML Files**

**1.1 Create index.html**

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Welcome to My Static Website</title>

<style>

body {

font-family: Arial, sans-serif;

text-align: center;

margin-top: 50px;

}

h1 {

color: #333;

}

</style>

</head>

<body>

<h1>Welcome to My Static Website!</h1>

<p>This is a simple static website hosted on Amazon S3. Enjoy exploring!</p>

</body>

</html>

**1.2 Create error.html**

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>404 - Page Not Found</title>

<style>

body {

font-family: Arial, sans-serif;

text-align: center;

margin-top: 50px;

}

h1 {

color: #f44336;

}

p {

color: #555;

}

</style>

</head>

<body>

<h1>404 - Page Not Found</h1>

<p>Sorry, the page you're looking for doesn't exist.</p>

<p>Please check the URL or <a href="/">go back to the homepage</a>.</p>

</body>

</html>

**Step 2: Module Configuration (modules/s3\_website)**

**2.1 Create main.tf**

resource "aws\_s3\_bucket" "terraformbucket" {

bucket = var.bucket\_name

}

resource "aws\_s3\_bucket\_ownership\_controls" "ownership" {

bucket = aws\_s3\_bucket.terraformbucket.id

rule {

object\_ownership = "BucketOwnerPreferred"

}

}

resource "aws\_s3\_bucket\_public\_access\_block" "publiceaccess" {

bucket = aws\_s3\_bucket.terraformbucket.id

block\_public\_acls = var.block\_public\_acls

block\_public\_policy = var.block\_public\_policy

ignore\_public\_acls = var.ignore\_public\_acls

restrict\_public\_buckets = var.restrict\_public\_buckets

}

resource "aws\_s3\_bucket\_acl" "bucket\_acl" {

depends\_on = [

aws\_s3\_bucket\_ownership\_controls.ownership,

aws\_s3\_bucket\_public\_access\_block.publiceaccess,

]

bucket = aws\_s3\_bucket.terraformbucket.id

acl = "public-read"

}

resource "aws\_s3\_object" "index" {

bucket = aws\_s3\_bucket.terraformbucket.id

key = "index.html"

source = var.index\_html

content\_type = "text/html"

}

resource "aws\_s3\_object" "error" {

bucket = aws\_s3\_bucket.terraformbucket.id

key = "error.html"

source = var.error\_html

content\_type = "text/html"

}

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

bucket = aws\_s3\_bucket.terraformbucket.id

index\_document {

suffix = "index.html"

}

error\_document {

key = "error.html"

}

}

resource "aws\_s3\_bucket\_policy" "public\_read\_access" {

depends\_on = [

aws\_s3\_bucket\_ownership\_controls.ownership,

aws\_s3\_bucket\_public\_access\_block.publiceaccess,

]

bucket = aws\_s3\_bucket.terraformbucket.id

policy = <<EOF

{

"Version": "2012-10-17",

"Statement": [

{

"Effect": "Allow",

"Principal": "\*",

"Action": [ "s3:GetObject" ],

"Resource": [

"${aws\_s3\_bucket.terraformbucket.arn}",

"${aws\_s3\_bucket.terraformbucket.arn}/\*"

]

}

]

}

EOF

}

**2.2 Create variables.tf**

variable "bucket\_name" {

description = "The name of the S3 bucket"

type = string

}

variable "index\_html" {

description = "Path to the index.html file to upload"

type = string

default = "index.html"

}

variable "error\_html" {

description = "Path to the error.html file to upload"

type = string

default = "error.html"

}

variable "block\_public\_acls" {

description = "Whether to block public ACLs on the bucket"

type = bool

default = false

}

variable "block\_public\_policy" {

description = "Whether to block public policy on the bucket"

type = bool

default = false

}

variable "ignore\_public\_acls" {

description = "Whether to ignore public ACLs on the bucket"

type = bool

default = false

}

variable "restrict\_public\_buckets" {

description = "Whether to restrict public buckets"

type = bool

default = false

}

**2.3 Create outputs.tf**

output "website\_endpoint" {

description = "The endpoint URL for the static website"

value = aws\_s3\_bucket.terraformbucket.website\_endpoint

}

**Step 3: Root Configuration**

**3.1 Create main.tf**

module "s3\_website" {

source = "./modules/s3\_website"

bucket\_name = "myteststaticbucketexample"

index\_html = "index.html"

error\_html = "error.html"

block\_public\_acls = false

block\_public\_policy = false

ignore\_public\_acls = false

restrict\_public\_buckets = false

}

output "website\_endpoint" {

value = module.s3\_website.website\_endpoint

}

**Step 4: Execute Terraform Commands**

**4.1 Initialize Terraform**

terraform init

**4.2 Plan Configuration**

terraform plan

**4.3 Apply Configuration**

terraform apply

**4.4 View Outputs**

terraform output website\_endpoint

**Step 5: Verification**

1. Access the S3 static website using the URL provided by the output.
2. Verify the index.html content is displayed.
3. Navigate to an invalid page to confirm the error.html response.

**Clean-Up**

1. Run terraform destroy to delete all resources:

terraform destroy

1. Verify that the S3 bucket and associated resources are deleted.

**Summary**

This lab covered:

* Creating a static website hosted on S3.
* Organizing Terraform code into reusable modules.
* Using variables for flexibility and outputs for communication between modules and root configurations.

**Terraform Import and Debugging Lab Guide**

**Part 1: Using Terraform Import**

**Step 1: Create a Resource First**

* Create a file named import.tf with the following content:

import {

id = "i-0430f7e44b388e61a"

to = aws\_instance.ec23

}

**Step 2: Import an Existing AWS Resource**

1. Execute the following command to generate the configuration for the imported resource:

terraform plan -generate-config-out=res.tf

1. Use the terraform import command to import an instance resource:

terraform import aws\_instance.web i-0c5b66a8f9d3b0053

1. Check the planned changes:

terraform plan

1. Apply the changes to save the imported resource state:

terraform apply

1. List the state resources to verify:

terraform state list

**Step 3: Import Multiple Resources**

* Update import.tf to include multiple resources:

import {

to = aws\_instance.web

id = "i-0c5b66a8f9d3b0053"

}

import {

to = aws\_s3\_bucket.bucket

id = "pradeep-tf-test-bucket"

}

* Generate configuration for a specific resource:

terraform plan -target=aws\_s3\_bucket.bucket -generate-config-out=s3.tf

* Import the S3 bucket resource:

terraform import aws\_s3\_bucket.bucket pradeep-tf-test-bucket

**Step 4: Manage State Resources**

* List resources in the state:

terraform state list

* Remove a specific resource from the state:

terraform state rm aws\_instance.web

* Remove all resources from the state:

terraform state rm '\*'

**Part 2: Debugging with Terraform Logs**

**Step 1: Enable Terraform Logs**

* Set the log level to TRACE:

export TF\_LOG=TRACE

* Other log levels include WARNING, ERROR, INFO, or VERBOSE.

**Step 2: Save Logs to a File**

* Specify the path to save logs:

export TF\_LOG\_PATH=/tmp/crash.log

**Step 3: Remove Terraform Logging**

* Unset the log level:

unset TF\_LOG

* Remove the log file path setting:

unset TF\_LOG\_PATH

**Notes:**

* Always verify the state list after importing resources to confirm successful addition.
* Use Terraform logs to troubleshoot issues or crashes efficiently.

**Example Workflow**

1. Import an EC2 instance and S3 bucket.
2. Use Terraform logs to debug potential issues during the import process.
3. Remove and re-import resources if necessary to test configurations.