**Terraform Lab Guide**

**1. Overview**

**Objective:** Understand Terraform’s primary purpose as a tool for infrastructure provisioning and compare it to configuration management tools like Ansible, Chef, Puppet, and SaltStack.

**2. Terminology**

* **Provider Block:** This is where you specify which provider (e.g., AWS, Azure, Google Cloud) you are using, along with authentication and region configurations. Example providers include:
  + AWS (Amazon Web Services)
  + Azure
  + Google Cloud Platform (GCP)
* **Resource Block:** This defines the resources to create. For instance, compute instances, databases, or firewalls. Resources describe the infrastructure you want.

**3. Purpose of Terraform**

Terraform is used primarily for **provisioning** infrastructure, including:

* Machines (e.g., EC2, GCP VMs)
* Databases (e.g., RDS, Cloud SQL)
* Networking (firewalls, VPCs, load balancers)

Terraform configurations are written in HashiCorp Configuration Language (**HCL**). It allows users to declaratively manage infrastructure as code (IaC).

**4. Differences Between Tools**

|  |  |  |
| --- | --- | --- |
| **Tool** | **Primary Focus** | **Secondary Focus** |
| Terraform | **Provisioning** | Infrastructure Automation |
| Ansible | **Configuration** | Limited Provisioning |
| Chef | **Configuration** | Limited Provisioning |
| Puppet | **Configuration** | Limited Provisioning |
| SaltStack | **Configuration** | Limited Provisioning |

* **Provisioning:** Creating resources (VMs, databases, etc.)
* **Configuration:** Modifying resources (e.g., installing packages, running commands)

**5. Interfaces for Managing Infrastructure**

|  |  |
| --- | --- |
| **Interface Type** | **Description** |
| **GUI (Graphical)** | Resources are created using cloud provider dashboards (manual method). |
| **CLI (Command Line)** | Administration tasks and resource provisioning done interactively. |
| **IaC (Infrastructure as Code)** | Automated creation using code with tools like Terraform. |

**6. Step-by-Step Lab Guide**

**Pre-Requisites**

1. Install Terraform: Follow the installation guide.
2. Set up an account with a cloud provider (AWS, Azure, GCP, etc.).
3. Install CLI tools for your provider (e.g., AWS CLI, Azure CLI).
4. Basic understanding of HCL syntax.

**Step 1: Initialize Terraform**

1. Create a folder named terraform-lab:

mkdir terraform-lab && cd terraform-lab

1. Create a Terraform configuration file named main.tf:

touch main.tf

**Step 2: Configure the Provider**

Edit main.tf with the following content:

**AWS Example:**

provider "aws" {

region = "us-east-1"

}

This block configures Terraform to use AWS as the cloud provider and sets the default region to us-east-1.

**Step 3: Add Resources**

Add the following aws\_instance resource to main.tf:

resource "aws\_instance" "example" {

ami = "ami-0c55b159cbfafe1f0" # Amazon Linux 2 AMI

instance\_type = "t2.micro"

tags = {

Name = "example-instance"

}

}

* This defines an EC2 instance with a specific AMI and instance type (t2.micro).

**Step 4: Initialize Terraform**

Run the following commands:

1. **Initialize the project:**

terraform init

1. Verify the output:
   * Downloads necessary provider plugins and initializes.

**Step 5: Plan Changes**

Generate an execution plan that describes the changes Terraform will make to your infrastructure:

terraform plan

* This ensures there are no unwanted changes before applying.

**Step 6: Apply Changes**

Execute the infrastructure creation process:

terraform apply

* Type yes when prompted.
* Observe Terraform provisioning the EC2 instance.

**Step 7: Verify Resources**

1. Login to your AWS console.
2. Navigate to **EC2 Instances**.
3. Verify that the example-instance appears.

**Step 8: Track Resource State**

Terraform keeps a state file (terraform.tfstate) to track resources. Use the following commands:

* View state:

terraform show

* List all resources:

terraform state list

**Step 9: Clean Up Resources**

1. Destroy all created resources using:

terraform destroy

1. Type yes to confirm the deletion.

**7. Summary of Terraform Concepts**

* **Provisioning vs. Configuration Tools:** Terraform is primarily for **provisioning**; use configuration tools like Ansible for managing configurations.
* **Provider Block:** Configures the cloud provider.
* **Resource Block:** Defines the actual infrastructure to be created.
* **State Management:** Tracks infrastructure using a state file.

**Next Steps**

Explore additional Terraform features:

1. Modules: Reusable code for infrastructure components.
2. Variables: Parameterize configurations for better reuse.
3. Remote Backends: Store your state in a remote backend like S3 or Terraform Cloud.

**Detailed Lab Guide: Setting Up a Centralized Terraform Machine**

**Objective**

Set up a centralized Terraform machine on an Ubuntu instance in the AWS region **nVirginia**, using the following:

* A t2.micro EC2 instance.
* An existing security group: **raman-sg**.
* A new SSH keypair.

**1. Setting Up the AWS Resources**

**Step 1: Configure the AWS Region**

Ensure your AWS CLI is configured for the nVirginia region (**us-east-1**):

aws configure

* **Region**: us-east-1
* **Access Key ID and Secret**: Input your AWS credentials.

**Step 2: Create a New Keypair**

Generate an SSH keypair to access the Terraform machine:

aws ec2 create-key-pair --key-name raman-keypair --query "KeyMaterial" --output text > raman-keypair.pem

chmod 400 raman-keypair.pem

* Save this file securely for SSH access.
* Use the file raman-keypair.pem while connecting to the instance.

**Step 3: Launch the EC2 Instance**

Launch an Ubuntu EC2 instance:

aws ec2 run-instances \

--image-id ami-04505e74c0741db8d \

--count 1 \

--instance-type t2.micro \

--key-name raman-keypair \

--security-groups raman-sg \

--tag-specifications 'ResourceType=instance,Tags=[{Key=Name,Value=raman-Cisco-centralizedtfMachine}]'

* **AMI**: Ubuntu 20.04 LTS (ensure it's valid in your region).
* **Security Group**: Reuse raman-sg.
* **Instance Name Tag**: raman-Cisco-centralizedtfMachine.

**Step 4: Verify the Instance**

Use the AWS CLI or console to ensure the instance is running:

aws ec2 describe-instances --filters "Name=tag:Name,Values=raman-Cisco-centralizedtfMachine"

Take note of the **Public IP** of your instance.

**2. Connecting to the Instance**

**Step 1: SSH into the EC2 Instance**

Use the saved keypair to access the instance:

ssh -i raman-keypair.pem ubuntu@<Public\_IP\_of\_Instance>

**Step 2: Switch to the Root User**

Once logged in, elevate privileges:

sudo -i

You’re now logged in as the root user on the Terraform centralized machine.

**3. Installing Terraform**

Follow HashiCorp’s official installation guide to set up Terraform.

**Step 1: Update System Packages**

Ensure all packages are up to date:

apt update && apt upgrade -y

**Step 2: Install Required Dependencies**

Terraform requires **curl** and **gnupg** to fetch and verify the installation package:

apt install -y gnupg software-properties-common curl

**Step 3: Add the HashiCorp GPG Key**

Add HashiCorp’s GPG key to verify the software package:

curl -fsSL https://apt.releases.hashicorp.com/gpg | gpg --dearmor > hashicorp.gpg

mv hashicorp.gpg /etc/apt/trusted.gpg.d/

**Step 4: Add HashiCorp’s Repository**

Add the Terraform repository to your system:

apt-add-repository "deb [signed-by=/etc/apt/trusted.gpg.d/hashicorp.gpg] https://apt.releases.hashicorp.com $(lsb\_release -cs) main"

**Step 5: Install Terraform**

Install Terraform using the package manager:

apt update && apt install terraform -y

**Step 6: Verify Installation**

Ensure Terraform is installed correctly:

terraform -v

Expected output:

Terraform v1.x.x

**4. Validating the Centralized Terraform Setup**

**Step 1: Initialize a Terraform Project**

1. Create a new directory for Terraform configurations:

mkdir /terraform-lab && cd /terraform-lab

1. Write a sample Terraform configuration file main.tf:
2. provider "aws" {
3. region = "us-east-1"
4. }
5. resource "aws\_s3\_bucket" "example" {
6. bucket = "example-terraform-bucket-${random\_id.id.hex}"
7. acl = "private"
8. }
9. resource "random\_id" "id" {
10. byte\_length = 8

}

**Step 2: Initialize Terraform**

Run the following command:

terraform init

**Step 3: Plan and Apply**

1. Preview the changes:

terraform plan

1. Apply the changes to create the resources:

terraform apply

1. Verify the output and confirm the S3 bucket creation.

**5. Summary and Next Steps**

* A centralized Terraform machine has been set up on AWS using a pre-existing security group, a new keypair, and an Ubuntu-based EC2 instance.
* Terraform was installed following HashiCorp’s guidelines.
* Resources can now be provisioned from this machine.

**Next Steps:**

* Extend the configuration to provision more complex infrastructures like VPCs, EC2 instances, or databases.
* Set up remote backends (e.g., S3, Terraform Cloud) to manage Terraform state files.

**1. Overview**

**Objective:** Provision a single AWS EC2 instance using Terraform and understand Terraform commands, provider configuration, and state management concepts.

**2. Key Terraform Commands**

1. **terraform init**
   * Installs the required providers and sets up the working directory for Terraform.
2. **terraform validate**
   * Validates the Terraform configuration files for syntax and logical errors.
3. **terraform plan**
   * Simulates the changes Terraform will apply to the environment. This is a dry run.
4. **terraform apply**
   * Applies the changes by making API calls to the provider specified in the provider block.
5. **terraform destroy**
   * Removes all resources defined in the Terraform configuration files.

**3. AWS Configuration and Authentication**

**AWS Credentials:**

* You can authenticate using the following methods:
  1. **Access Key and Secret Key:** Specify them directly in the provider block.
  2. **Environment Variables:** Set AWS\_ACCESS\_KEY\_ID and AWS\_SECRET\_ACCESS\_KEY in your environment.
  3. **IAM Roles:** Use a role assigned to your EC2 instance for authentication.

**4. Desired State and State Management**

* **Desired State:** Defined in Terraform HCL (HashiCorp Configuration Language). Example:

instance\_type = "t2.micro"

* **Terraform State File (terraform.tfstate):** Tracks the infrastructure’s last applied state.
* **Actual/Current State:** The live state of resources in the cloud provider's environment.

**Example Workflow:**

1. Desired State: Instance type is t2.micro.
2. Modify Desired State to t2.medium.
3. Run terraform apply: Terraform updates the instance from t2.micro → t2.medium → t2.micro.

**5. Lab Details**

**AWS Region and Availability Zone:**

* **Region:** us-east-1
* **Availability Zone:** Determined automatically by AWS.

**Resource Details:**

1. **Resource Name:** raman-first-server
2. **Instance Type:** t2.micro
3. **AMI (Amazon Machine Image):** ami-0e2c8caa4b6378d8c
4. **Tags:** Adds metadata like the instance name.
5. **Public IP and Private IP:** Assigned automatically by AWS.

**6. Step-by-Step Instructions**

**Step 1: Create the Terraform Configuration File**

1. Log in to your central Terraform machine.
2. Create a file named first.tf and include the following content:

# Configure the AWS Provider

provider "aws" {

region = "us-east-1"

access\_key = "AKIAZ7FSO3B5YFZTT5QT"

secret\_key = "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"

}

# Define the AWS Instance Resource

resource "aws\_instance" "ec2" {

ami = "ami-0e2c8caa4b6378d8c"

instance\_type = "t2.micro"

tags = {

Name = "raman-first-server"

}

}

**Step 2: Initialize Terraform**

* Run terraform init to install AWS provider plugins and prepare the working directory.

**Step 3: Validate the Configuration**

* Run terraform validate to ensure that the configuration is syntactically and logically correct.

**Step 4: Plan the Deployment**

* Run terraform plan to view the infrastructure changes Terraform will apply.

**Step 5: Apply the Configuration**

* Run terraform apply to create the EC2 instance.
* Confirm the execution when prompted by typing yes.

**Step 6: Verify the Instance**

* Log in to your AWS Management Console.
* Navigate to the **EC2 Dashboard** and verify:
  1. Instance Name: raman-first-server
  2. Instance Type: t2.micro
  3. Region: us-east-1

**Step 7: Modify the Desired State**

1. Change the instance type in first.tf from t2.micro to t2.medium.
2. Run terraform apply again to update the instance.
3. Verify that the instance type has changed in AWS.

**Step 8: Destroy the Resources**

* Run terraform destroy to remove the EC2 instance and clean up resources.
* Confirm the execution when prompted by typing yes.

**7. Summary**

* Terraform provisions resources (EC2 instances in this lab) using code.
* Desired state is defined in the configuration file (first.tf).
* Changes are tracked and applied based on the Terraform state file.
* Modify configurations easily and rerun terraform apply to update the infrastructure.

**Terraform Lab Guide: Advanced EC2 Instance Management**

**1. Overview**

This lab demonstrates advanced EC2 instance management using Terraform. In addition to provisioning an EC2 instance, this guide introduces the use of the aws\_ec2\_instance\_state resource to control and track the instance's desired state (e.g., running or stopped).

**Objective:**

* Provision an EC2 instance.
* Specify and manage the instance state using Terraform.

**2. Key Concepts**

1. **AWS Provider Configuration:**
   * Specifies AWS region and authentication credentials.
2. **Resource Management:**
   * EC2 instance is created and managed with desired configurations like instance type, AMI, and tags.
3. **Instance State Management:**
   * Use aws\_ec2\_instance\_state to ensure the instance is in a specific state (e.g., running or stopped).
4. **Critical Arguments in Desired State:**
   * Always define critical arguments (e.g., instance\_type, ami, vpc\_security\_group\_ids) in the desired state for predictable behavior.

**3. Environment Details**

* **Region:** us-east-1
* **Resource Name:** raman-first-server
* **Instance Type:** t2.micro
* **AMI (Amazon Machine Image):** ami-0e2c8caa4b6378d8c
* **Security Group ID:** sg-0e8552c15babe88bc (replace with your existing security group ID if different).

**4. Terraform Configuration**

**Step 1: Provider Configuration**

Ensure AWS credentials and region are set in the provider block:

provider "aws" {

region = "us-east-1"

access\_key = "AKIAZ7FSO3B5YFZTT5QT"

secret\_key = "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"

}

**Step 2: EC2 Instance Definition**

Define the EC2 instance resource with the required parameters:

resource "aws\_instance" "ec2" {

ami = "ami-0e2c8caa4b6378d8c"

instance\_type = "t2.micro"

vpc\_security\_group\_ids = ["sg-0e8552c15babe88bc"]

tags = {

Name = "raman-first-server"

}

}

**Step 3: Instance State Management**

Use aws\_ec2\_instance\_state to enforce the desired state (e.g., running or stopped):

resource "aws\_ec2\_instance\_state" "ec22" {

instance\_id = aws\_instance.ec2.id

state = "running" # Change to "stopped" if necessary

}

**5. Implementation Steps**

**Step 1: Initialize Terraform**

* Run terraform init to download required plugins and prepare the environment.

**Step 2: Validate Configuration**

* Use terraform validate to ensure correctness of the configuration.

**Step 3: Plan Changes**

* Run terraform plan to preview the resources to be created or modified.

**Step 4: Apply Configuration**

* Execute terraform apply to provision the EC2 instance and enforce the desired state.

**Step 5: Verify Resources**

1. Log in to the AWS Management Console.
2. Navigate to **EC2 Dashboard** and check:
   * Instance Name: raman-first-server
   * Instance State: running
   * Tags, Security Group, Instance Type.

**6. Modifications and State Updates**

**Example: Changing Instance Type**

1. Modify the instance type in the aws\_instance resource block:

instance\_type = "t2.medium"

1. Run terraform plan and then terraform apply to update the instance type.

**Example: Stopping the Instance**

1. Modify the aws\_ec2\_instance\_state resource:

state = "stopped"

1. Run terraform apply to stop the EC2 instance.

**7. Best Practices**

* Always define **critical arguments** (e.g., instance\_type, ami, vpc\_security\_group\_ids, etc.) to ensure reliable infrastructure management.
* Use .tfvars files or environment variables to manage sensitive information like access keys.
* Regularly review the terraform.tfstate file for accuracy.
* Keep configuration files under version control for auditing and rollback purposes.

**8. Cleanup**

* Use terraform destroy to delete all provisioned resources when they are no longer required.

**Terraform Lab Guide: Creating a GitHub Repository**

This lab demonstrates how to use Terraform to manage GitHub repositories. We will go through setting up the GitHub provider, creating a repository, initializing the Terraform configuration, and working with various Terraform commands to manage the state of the resources.

**Prerequisites:**

* A GitHub account with access to an API token.
* Terraform installed on your machine.
* A basic understanding of how Terraform works.

**1. Setting up the Terraform Configuration**

* Open a terminal in your chosen working directory.
* Create a file named git.tf which will contain the provider configuration and the resource definition for the repository.

bash

Copy code

# Create the git.tf file using any editor (e.g., vi or nano)

vi git.tf

**Contents of git.tf file:**

hcl

Copy code

provider "github" {

token = "ghp\_vOU5C7OFC6y5ubrjuZq23SOHlNBf9s1sY4BI" # Replace with your actual GitHub token

}

resource "github\_repository" "ex" {

name = "example" # The repository name

description = "My awesome codebase" # The description of the repository

visibility = "public" # Set visibility to public or private

}

In this step, the provider is configured to authenticate using a GitHub token, and a GitHub repository resource (github\_repository.ex) is defined with some attributes like name, description, and visibility.

**2. Initialize Terraform**

Run the following command to initialize your Terraform working directory and download the necessary provider plugins:

bash

Copy code

terraform init -upgrade

This step will automatically download the GitHub provider specified in your configuration. The -upgrade flag ensures that Terraform gets the latest available version.

**3. Check Terraform Configuration**

After initializing, you can check the status of the configuration by listing the files and verifying the .terraform.lock.hcl file, which contains provider versions:

bash

Copy code

cat .terraform.lock.hcl

**4. View and Validate the Plan**

Before applying the configuration to create resources, you should preview the changes that will be applied. Use the terraform plan command:

bash

Copy code

terraform plan

This command checks your configuration files and provides an execution plan, showing which actions Terraform will perform (e.g., creating a new GitHub repository).

**5. Apply the Terraform Plan**

Once you've confirmed the plan, run the terraform apply command to apply the changes and create the GitHub repository:

bash

Copy code

terraform apply

Terraform will prompt for confirmation before proceeding. Type yes to proceed with the creation of the repository.

**6. Check the State File**

To view the state file, which keeps track of your resources, use:

bash

Copy code

cat terraform.tfstate

The state file contains the JSON data on your infrastructure.

**7. Listing Resources in Terraform**

To list the resources managed by Terraform, use:

bash

Copy code

terraform state list

This will show a list of resources like github\_repository.ex that have been created and are tracked by Terraform.

**8. Destroy the Resource**

If you need to destroy the created GitHub repository (perhaps for testing), use the following command:

bash

Copy code

terraform destroy -target=github\_repository.ex

The -target option ensures that only the specified resource (in this case, github\_repository.ex) will be destroyed.

**9. Verify State After Destruction**

Once the resource has been destroyed, you can check the state again to verify that the resource has been removed:

bash

Copy code

terraform state list

You should see that the github\_repository.ex is no longer listed.

**Command Summary:**

| **Command** | **Description** |
| --- | --- |
| vi git.tf | Create or edit the git.tf configuration file |
| terraform init -upgrade | Initialize the Terraform working directory and provider |
| cat .terraform.lock.hcl | View the lock file to ensure the provider version is correct |
| terraform plan | Preview the Terraform execution plan |
| terraform apply | Apply the Terraform plan to create the repository |
| terraform state list | List all Terraform-managed resources |
| cat terraform.tfstate | View the raw state file |
| terraform destroy | Destroy resources (e.g., repository) |

**Common Errors and Troubleshooting:**

* **Error: Invalid token:** Ensure that the GitHub API token provided is valid and has the necessary permissions.
* **Error: No such file:** If you encounter errors like "No such file," double-check your directory structure and verify that Terraform is running from the directory containing your .tf configuration files.
* **Missing Resource in State:** If resources are missing or not applied properly, you can try running terraform plan again or check the logs for more information.

This completes the basic process for managing a GitHub repository using Terraform, from configuration to resource destruction.