Practical Lab Guide 2: Automating Infrastructure Provisioning with Terraform

Introduction:

In this lab, you'll set up the foundational components of a web application, including the network, compute resources, and backend storage for state management. Using Terraform, you'll deploy key infrastructure elements, ensuring scalability and security. This exercise demonstrates how to manage and automate cloud infrastructure effectively.

Step 1: Installing Terraform on Windows

1. Download Terraform:
   - Visit the Terraform Downloads Page for Windows.
   - Click on the appropriate link to download the Terraform executable (.exe) file for Windows.
2. Install Terraform:
   - Navigate to the directory where you downloaded the Terraform executable.
   - Cut or copy the terraform.exe file.
   - Paste it into the C:\Program Files\terraform directory. (You might need administrative privileges to do this.)
3. Set Up the System Path:
   - Open the Start Search, type System and select System Control Panel.
   - Click on Advanced system settings and then Environment Variables.
   - Under System Variables, find and select the Path variable, then click Edit.
   - Click New and add the path C:\Program Files\terraform.
   - Click OK to close all dialogs.
4. Verify Installation:

5. Open Command Prompt and type:

```
terraform -version
```

   - This command should return the Terraform version if it was installed correctly.

Installing Terraform on Linux /Ubuntu

1: Configure packages

- We will install the pre-build versions of Terraform using apt package manager.
- We will first start by downloading and saving the hashicorp PGP keys:

  -
    ```
    wget -O- https://apt.releases.hashicorp.com/gpg | sudo gpg --dearmor
    -o /usr/share/keyrings/hashicorp-archive-keyring.gpg
    ```

- We can now add an entry to the system's list of package providers:
  - `echo "deb [signed-by=/usr/share/keyrings/hashicorp-archive-keyring.gpg] https://apt.releases.hashicorp.com $(lsb_release -cs) main" | sudo tee /etc/apt/sources.list.d/hashicorp.list`

2: Install Terraform

- We can now update our package list and install Terraform:
  - `sudo apt update && sudo apt install terraform`

Run terraform -version to test if Terraform has been correctly installed. This will show you the Terraform version installed:

3: Setting Up AWS CLI

1. Create AWS IAM User and Access Key:
   - Log into the AWS Management Console.
   - Navigate to IAM (Identity and Access Management).
   - Go to Users and select Your name.
   - On Security credentials select Create Access key
   - On the final screen, you'll see the Access key ID and Secret access key. Download or copy these credentials securely.
2. Configure AWS CLI:
   - Open Command Prompt on your local machine.
   - Type aws configure and press Enter.
   - Enter the Access key ID and Secret access key when prompted.
   - For Default region name, enter the AWS Region (e.g., us-east-1).
   - For Default output format, enter json (or you can leave it as default if unsure).
3. Verify AWS CLI Configuration:
   - The credentials and config file are updated when you run aws configure.
   - The config file is located at C:\Users\USERNAME\.aws\config on Windows.

4. You can check if AWS CLI is configured properly by running a test command, e.g., bash: aws s3 ls

   - If the setup is correct, this command should list the S3 buckets available in your AWS account

Step 2: Create a New Project Directory

1. First, create a directory for your Terraform project and navigate into it.

```
mkdir my-terraform-project
cd my-terraform-project
mkdir remote-backend
```

Create a folder `remote-backend` to store the Terraform state management resources (S3 and DynamoDB)

Four files to be created :

- **`main.tf`**: This is the main configuration file that defines the AWS provider, the S3 bucket (for state storage), versioning for the bucket, encryption, and the DynamoDB table (for state locking).
- **`variables.tf`**: This file defines the input variables for your Terraform configuration, such as the names of the S3 bucket and DynamoDB table.
- **`outputs.tf`**: This outputs the S3 bucket ARN and DynamoDB table name after Terraform provisions the resources.
- **`terraform.tfvars`**: This optional file can be used to provide specific values for variables.

2. Create the Main Configuration File (`main.tf`) inside (`remote-backend`) folder

This file will contain your Terraform configuration and resource definitions.

Create a Terraform configuration without a remote backend (defaults to a **local backend**)

# main.tf

```
terraform {

  required_providers {

    aws = {

      source  = "hashicorp/aws"

      version = "~> 3.0"

    }

  }

}
```

This block specifies the providers that Terraform requires. In this case, it defines the AWS provider, indicating that Terraform should use the AWS provider from HashiCorp, and it locks the version to approximately version 3.0, ensuring compatibility and stability in the provisioned resources.

# AWS provider configuration

```
provider "aws" {

  region = "us-east-1"
}
```

This block configures the AWS provider, setting the AWS region to us-east-1. This is the region where all resources in this Terraform configuration will be created unless otherwise specified in individual resource blocks.

Define the necessary AWS resources: S3 bucket and DynamoDB table with a hash key named "**LockID**"

# S3 bucket for storing Terraform state

```
resource "random_id" "bucket_suffix" {

 byte_length = 4

}

terraform {

  required_providers {

    aws = {

      source  = "hashicorp/aws"

      version = "~> 3.0"

    }

    random = {

      source  = "hashicorp/random"

      version = "~> 3.0"  # Specify a version constraint

    }

  }

}
```

to ensure the uniqueness of the bucket name

```
resource "aws_s3_bucket" "terraform_state" {

  bucket        = "${var.s3_bucket_name}-${random_id.bucket_suffix.hex}"

  force_destroy = true

}
```

# Enable versioning for the S3 bucket

```
resource "aws_s3_bucket_versioning" "terraform_bucket_versioning" {

  bucket = aws_s3_bucket.terraform_state.id

  versioning_configuration {

    status = "Enabled"

  }

}
```

This block enables versioning on the S3 bucket used for storing Terraform state. Versioning is crucial for backup and recovery, as it allows you to preserve, retrieve, and restore every version of every object stored in the S3 bucket.

# Enable server-side encryption for the S3 bucket

```
resource "aws_s3_bucket_server_side_encryption_configuration"
"terraform_state_crypto_conf" {

  bucket = aws_s3_bucket.terraform_state.bucket

  rule {

    apply_server_side_encryption_by_default {

      sse_algorithm = "AES256"

    }

  }

}
```

This block configures server-side encryption for the S3 bucket using the AES256 encryption algorithm. It ensures that all objects stored in the bucket are encrypted at rest, enhancing the security of your stored Terraform state.

# DynamoDB table for state locking

```
resource "aws_dynamodb_table" "terraform_locks" {

  name         = var.dynamodb_table_name

  billing_mode = "PAY_PER_REQUEST"

  hash_key     = "LockID"
  attribute {

    name = "LockID"

    type = "S"

  }

}
```

This block creates a DynamoDB table with a primary key named LockID of type String. The table is used for locking the Terraform state to prevent concurrent executions of Terraform that could lead to state corruption. The PAY_PER_REQUEST billing mode is used to optimize costs, making this solution cost-effective for environments where state locking operations are infrequent.

3. Create the Variables File (`variables.tf`)

Define the variables used in the `main.tf` configuration. These will make your Terraform configuration more flexible and avoid hard coding variables.

# variables.tf

# S3 Bucket Name

```
variable "s3_bucket_name" {

  description = "Name of the S3 bucket to store Terraform state"

  type        = string

  default     = "enis-terraform-for-state-file-0125"

}
```

# DynamoDB Table Name

```
variable "dynamodb_table_name" {

  description = "Name of the DynamoDB table for state locking"

  type        = string

  default     = "terraform-state-locking-1256"
}
```

4. Create the Outputs File (`outputs.tf`)

You can output useful information after the infrastructure is provisioned.

# outputs.tf

# S3 Bucket ARN

```
output "s3_bucket_arn" {

  description = "ARN of the S3 bucket for Terraform state"

  value       = aws_s3_bucket.terraform_state.arn

}
```

ARN stands for **Amazon Resource Name**. It's a globally unique identifier used by AWS to identify resources.

# DynamoDB Table Name

```
output "dynamodb_table_name" {

  description = "Name of the DynamoDB table for state locking"

  value       = aws_dynamodb_table.terraform_locks.name

}
```

The **DynamoDB table** manages **state locking** to prevent concurrent modifications of the **Terraform state** in the **S3 bucket**.

DynamoDB Table (with S3 Backend): Manages concurrent access to the **state file** stored in the S3 bucket, ensuring that only one person or process can modify the infrastructure at a time.

5. Create the Terraform Variables File (`terraform.tfvars`)

This is where you provide specific values for the variables you've defined. You can leave this file empty if you're using default values from the `variables.tf` file.

# terraform.tfvars

# Optionally, you can override the default values

`s3_bucket_name      = "custom-terraform-state-bucket-123456"` # it should be unique

`dynamodb_table_name = "custom-terraform-state-locks-123456"` # it should be unique

6. Initialize Terraform

Before running any Terraform commands, you need to initialize the project. This downloads the necessary provider plugins.

```
cd remote_backend
```

```
terraform init
```

7. Run a Terraform Plan

The `terraform plan` command allows you to see what Terraform will do when you apply the configuration.

Review the output to ensure Terraform is creating the expected resources.

8. Apply the Terraform Configuration

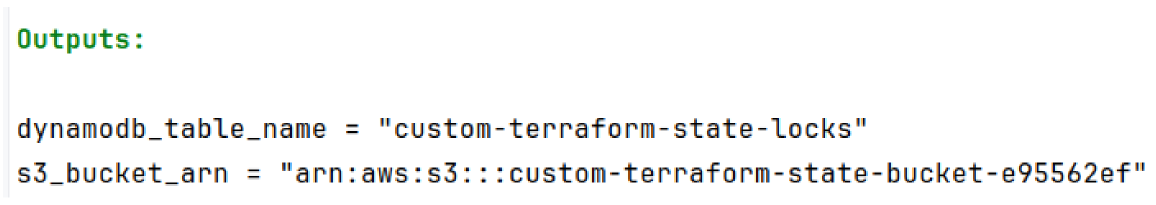
Run the `terraform apply` command to create the S3 bucket and DynamoDB table. Terraform will prompt you to confirm the changes.

`terraform apply` ,Type `yes` to apply the changes.

The `.terraform.lock.hcl` file is a lock file created by Terraform that helps ensure consistency and stability when managing infrastructure. Here's why it matters:

- **Locks Provider Versions**: The `.terraform.lock.hcl` file stores specific versions of the providers (such as AWS, Random, etc.) that your Terraform project uses. This ensures that every time you run Terraform commands, the same versions of providers are used, avoiding unexpected changes or incompatibility issues caused by provider updates.

9. Verify the Outputs



```
Outputs:

dynamodb_table_name = "custom-terraform-state-locks"
s3_bucket_arn = "arn:aws:s3:::custom-terraform-state-bucket-e95562ef"
```

After the apply is complete, Terraform will output the specified values such as the **S3 Bucket ARN** and **DynamoDB Table Name**.

10. Update the Terraform Configuration with Remote Backend

In this part of the lab, you will update your Terraform configuration to use a **remote backend** to store the Terraform state in an S3 bucket and use a DynamoDB table for state locking. This is a common approach to centralize state management, especially in team environments.

now the `main.tf` file to include the `backend` configuration for remote state management. The `backend "s3"` block will ensure that Terraform stores the state file in the S3 bucket and uses the DynamoDB table to prevent concurrent state modifications.

# Remote backend configuration using S3 and DynamoDB for state locking

```
terraform {

 backend "s3" {

   bucket= "custom-terraform-state-bucket-e95562ef"  # Replace with your S3 bucket name

   key = "aws-backend/terraform.tfstate" # Location of the state file in the bucket

   region = "us-east-1" # AWS region

     dynamodb_table = "custom-terraform-state-locks" # Replace with your DynamoDB table
name

   encrypt = true  # Enables encryption for the state file

 }

}
```

11. Re-initialize Terraform to Use the Remote Backend

Since you've updated the backend configuration, you need to reinitialize Terraform so that it connects to the remote backend instead of the local state.

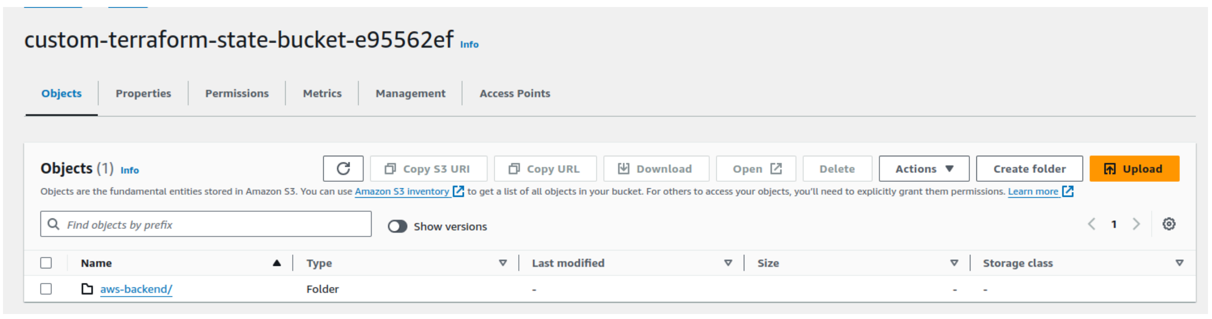
```
terraform init
```

After re-initializing Terraform, you can proceed with the usual workflow to review the changes and apply them.

12. Verify Remote State in S3 and DynamoDB

After running the `apply` command, you can verify that Terraform has successfully created the resources and that the state file is stored in the S3 bucket.

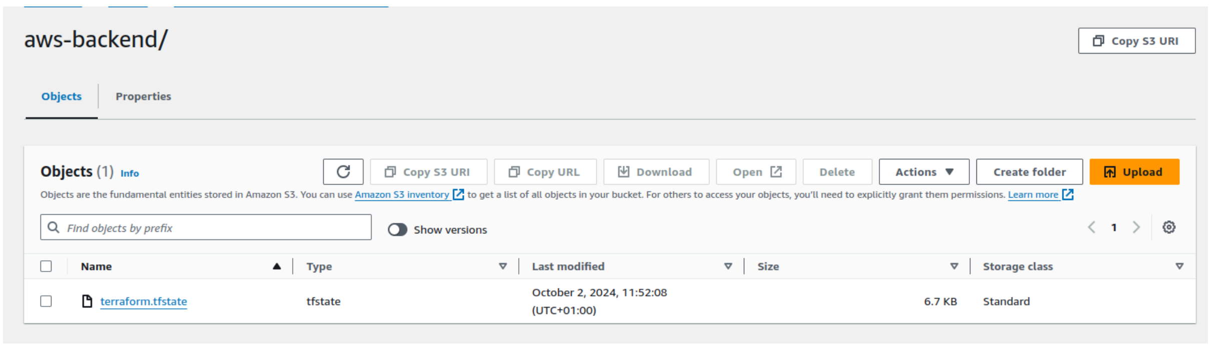
- **S3 Bucket**: Navigate to the S3 bucket in the AWS Console, and you should see a file named `terraform.tfstate` under the path you specified (`aws-backend/terraform.tfstate`).
- **DynamoDB Table**: In the AWS DynamoDB console, look for the table `terraform-state-locking-1256` and ensure that locks are being created when Terraform runs.


custom-terraform-state-bucket-e95562ef Info

Objects (1) Info

Objects are the fundamental entities stored in Amazon S3. You can use Amazon S3 inventory to get a list of all objects in your bucket. For others to access your objects, you'll need to explicitly grant them permissions. Learn more

Name	Type	Last modified	Size	Storage class
aws-backend/	Folder	-	-	-

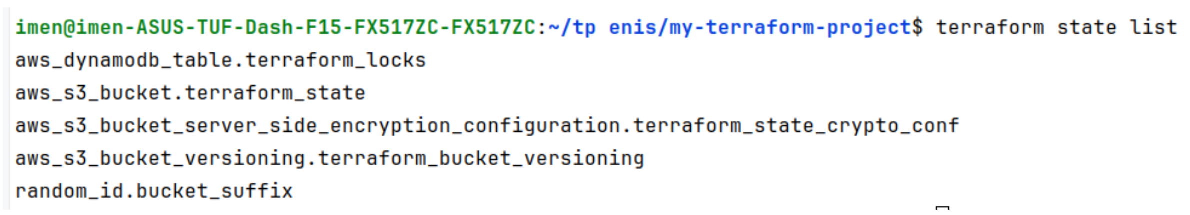

aws-backend/

Objects (1) Info

Objects are the fundamental entities stored in Amazon S3. You can use Amazon S3 inventory to get a list of all objects in your bucket. For others to access your objects, you'll need to explicitly grant them permissions. Learn more

Name	Type	Last modified	Size	Storage class
terraform.tfstate	tfstate	October 2, 2024, 11:52:08 (UTC+01:00)	6.7 KB	Standard

`terraform state list`: To list all the resources in the state.


```
imen@imen-ASUS-TUF-Dash-F15-FX517ZC-FX517ZC:~/tp enis/my-terraform-project$ terraform state list
aws_dynamodb_table.terraform_locks
aws_s3_bucket.terraform_state
aws_s3_bucket_server_side_encryption_configuration.terraform_state_crypto_conf
aws_s3_bucket_versioning.terraform_bucket_versioning
random_id.bucket_suffix
```

13. Creation of network components

Now we will move to our main folder since the state file is set up remotely and we will create same 3 files (main.tf , variables.tf and output.tf) for the app hosting infrastructure

```
cd ..
terraform init
```

Overview of Network Components

The network infrastructure includes several essential components that work together to create a secure and scalable architecture on AWS. These components are:

1. **VPC (Virtual Private Cloud)**:
   - Acts as the primary container for your networking resources. It's a logically isolated section of the AWS cloud.
   - **Purpose**: Provides control over networking aspects like IP addressing, routing, and security.
2. **Subnets**:
   - Logical subdivisions within your VPC that allow you to separate resources into public and private zones.
   - **Purpose**: Helps segregate resources and provide security by separating public-facing resources from private ones.
3. **Internet Gateway**:
   - A gateway that connects your VPC to the internet.
   - **Purpose**: Allows instances in public subnets to communicate with the internet.
4. **Route Tables**:
   - Defines how traffic is directed in and out of your subnets.
   - **Purpose**: Allows traffic routing between the internet and subnets or between different subnets.
5. **NAT Gateway (Optional)**:
   - Enables instances in private subnets to connect to the internet without being directly exposed to it.
   - **Purpose**: Provides outbound internet access for instances in private subnets without exposing them to the public internet.

Relationship Between Components:

- The **VPC** acts as the container that hosts all other network components.
- **Subnets** are created inside the VPC to define logical partitions.
- The **Internet Gateway** enables public subnets to communicate with the internet.
- **Route tables** ensure that traffic from the subnets is routed correctly to the internet or other destinations.
- The **NAT Gateway** allows private subnets to access the internet while remaining private.

Now let's build the network infrastructure, while giving a brief description of each block:

VPC Creation (aws_vpc):

Defines a Virtual Private Cloud (VPC) with a CIDR block of 10.0.0.0/16, enabling DNS support and hostnames for resources within the VPC. It's tagged as "tp_cloud_devops_vpc".

```
resource "aws_vpc" "tp_cloud_devops_vpc" {

  cidr_block= var.vpc_cidr_block # Using variable for VPC CIDR

  enable_dns_support   = true

  enable_dns_hostnames = true

  tags = {

    Name = "tp_cloud_devops_vpc"

  }

}
```

Subnet Creation (aws_subnet for public_subnet_1):

Creates a subnet within the VPC defined above, specifying a CIDR block of 10.0.1.0/24 in the us-east-1a availability zone. It enables public IP assignment on instance launch and is tagged as "PublicSubnet1".

```
resource "aws_subnet" "public_subnet_1" {

  vpc_id            = aws_vpc.tp_cloud_devops_vpc.id

  cidr_block        = var.public_subnet_1_cidr

  availability_zone = var.availability_zone_1

  map_public_ip_on_launch = true

  tags = {

    Name = "PublicSubnet1"

  }

}
```

Subnet Creation (aws_subnet for public_subnet_2):

Similar to the first subnet but located in a different availability zone (us-east-1b) with a CIDR block of 10.0.2.0/24. It's also configured for public IP assignment on launch and tagged as "PublicSubnet2".

```
resource "aws_subnet" "public_subnet_2" {

  vpc_id            = aws_vpc.tp_cloud_devops_vpc.id

  cidr_block        = var.public_subnet_2_cidr

  availability_zone = var.availability_zone_2

  map_public_ip_on_launch = true

  tags = {

    Name = "PublicSubnet2"

  }

}
```

Internet Gateway Creation (aws_internet_gateway):

Attaches an internet gateway to the VPC to allow communication between resources in the VPC and the internet. It is tagged as "MainInternetGateway".

```
resource "aws_internet_gateway" "main_gateway" {

  vpc_id = aws_vpc.tp_cloud_devops_vpc.id

  tags = {

    Name = "MainInternetGateway"

  }

}
```

Route Table Creation (aws_route_table):

Establishes a route table for the VPC with a route that directs all traffic (denoted by 0.0.0.0/0) to the internet gateway, facilitating internet access for the VPC. This table is tagged as "PublicRouteTable".

```
resource "aws_route_table" "public_route_table" {

  vpc_id = aws_vpc.tp_cloud_devops_vpc.id

  route {

    cidr_block = "0.0.0.0/0"

    gateway_id = aws_internet_gateway.main_gateway.id
  }

  tags = {

    Name = "PublicRouteTable"

  }

}
```

Route Table Association (aws_route_table_association for public_rta1 and public_rta2):

Associates the first subnet (public_subnet_1) and the second subnet (public_subnet_2) with the public route table, ensuring that both subnets can route traffic to and from the internet through the internet gateway.

```
resource "aws_route_table_association" "public_rta1" {

  subnet_id      = aws_subnet.public_subnet_1.id

  route_table_id = aws_route_table.public_route_table.id

}

resource "aws_route_table_association" "public_rta2" {

  subnet_id      = aws_subnet.public_subnet_2.id

  route_table_id = aws_route_table.public_route_table.id

}
```

In new file variables.tf in main initial folder

Now, let's make sure that the variables used in the main.tf are defined properly in the variables.tf file.

# VPC CIDR Block

```
variable "vpc_cidr_block" {

  description = "The CIDR block for the VPC"

  type        = string

  default     = "10.0.0.0/16"

}
```

# Public Subnet 1 CIDR Block

```
variable "public_subnet_1_cidr" {

  description = "CIDR block for public subnet 1"

  type        = string

  default     = "10.0.1.0/24"

}
```

# Public Subnet 2 CIDR Block

```
variable "public_subnet_2_cidr" {

  description = "CIDR block for public subnet 2"

  type        = string

  default     = "10.0.2.0/24"

}
```

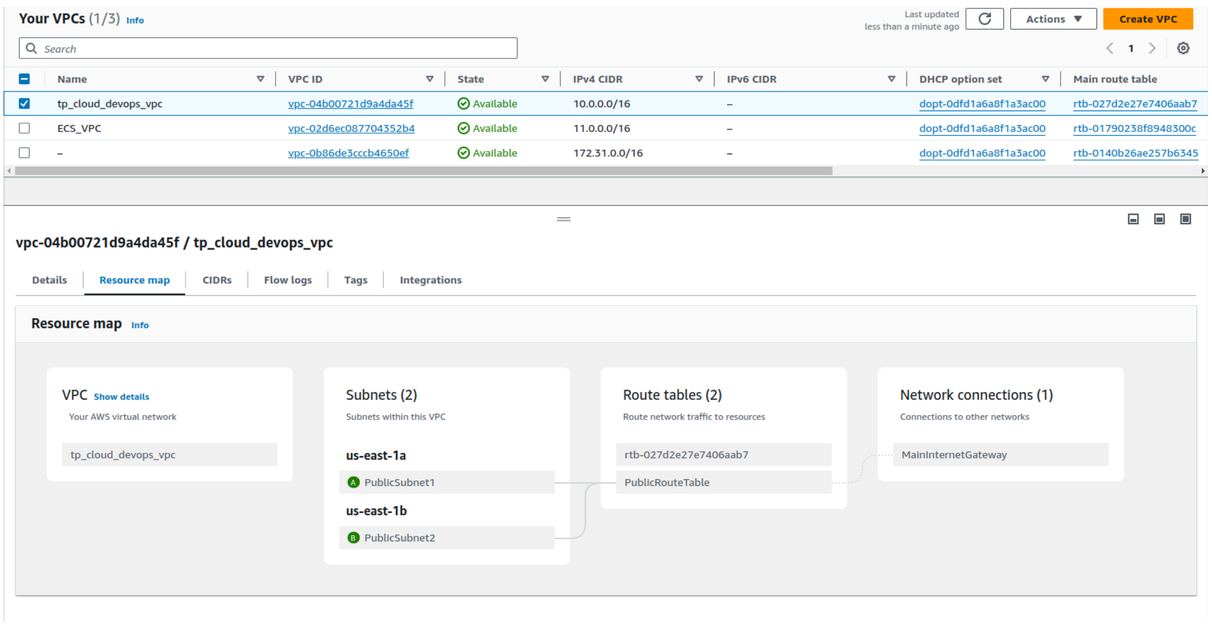
# Availability Zone for Subnet 1

```
variable "availability_zone_1" {

  description = "Availability zone for the first public subnet"

  type        = string

  default     = "us-east-1a"

}
```

# Availability Zone for Subnet 2

```
variable "availability_zone_2" {

  description = "Availability zone for the second public subnet"

  type        = string

  default     = "us-east-1b"

}
```

Run terraform plan then apply and check changes in the console:


Your VPCs (1/3) Info

Last updated less than a minute ago

	Name	VPC ID	State	IPv4 CIDR	IPv6 CIDR	DHCP option set	Main route table
✓	tp_cloud_devops_vpc	vpc-04b00721d9a4da45f	Available	10.0.0.0/16	–	dopt-0dfd1a6a8f1a3ac00	rtb-027d2e27e7406aab7
	ECS_VPC	vpc-02d6ec087704352b4	Available	11.0.0.0/16	–	dopt-0dfd1a6a8f1a3ac00	rtb-01790238f8948300c
	–	vpc-0b86de3cccb4650ef	Available	172.31.0.0/16	–	dopt-0dfd1a6a8f1a3ac00	rtb-0140b26ae257b6345

vpc-04b00721d9a4da45f / tp_cloud_devops_vpc

Resource map Info

VPC Show details
Your AWS virtual network
tp_cloud_devops_vpc

Subnets (2)
Subnets within this VPC
us-east-1a
A PublicSubnet1
us-east-1b
B PublicSubnet2

Route tables (2)
Route network traffic to resources
rtb-027d2e27e7406aab7
PublicRouteTable

Network connections (1)
Connections to other networks
MainInternetGateway

14. Creation of compute components

In this section of the lab, we'll build the compute infrastructure, focusing on creating an EC2 instance, SSH key pairs, security groups, and launching a simple web server. We will also update our network configuration and make use of variables for flexibility.

Overview of Components

1. **TLS Private Key:** We generate an RSA private key for SSH access to our EC2 instance.
2. **AWS Key Pair:** Using the public key from the generated RSA key pair, we create an AWS key pair for secure access.
3. **S3 Bucket:** We securely store the generated SSH private key in an S3 bucket for future use.
4. **EC2 Instance:** An Amazon EC2 instance is launched with the generated key pair, running a simple Python HTTP server.
5. **Security Group:** A security group is created to allow inbound traffic for HTTP (port 8080) and SSH (port 22) access.

Update main.tf file to add the compute components

# Generate an SSH key pair using TLS provider

```
resource "tls_private_key" "example_ssh_key" {
  algorithm = "RSA"
  rsa_bits  = 2048
}
```

This resource generates an RSA private key with a key size of 2048 bits. It's used to create cryptographic keys that can securely manage encryption, decryption, and authentication processes

Add the TLS Provider to Your Configuration

If you haven't explicitly defined the `tls` provider in your Terraform configuration, you need to add it in the `terraform` block in file **main.tf** in folder **`remote-backend`**

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 3.0"
    }
    random = {
      source  = "hashicorp/random"
      version = "~> 3.0"
    }
    tls = {
      source  = "hashicorp/tls"
      version = "~> 3.0"
    }
  }
}
```

Run `terraform init -upgrade`

# Create an AWS Key Pair using the public key from the TLS private key

```
resource "aws_key_pair" "deployer_key" {
  key_name    = var.ssh_key_name
  public_key = tls_private_key.example_ssh_key.public_key_openssh
}
```

This block creates an AWS key pair using the public key generated by the tls_private_key resource. The key pair is identified by the name deployer_key in AWS, and it allows you to access AWS instances securely via SSH.

# Store the SSH private key in an S3 bucket for secure storage (we can use same bucket of the state)

# Upload the private key to the S3 bucket

```
resource "aws_s3_object" "private_key_object" {
  bucket                  =  # Reference existing S3 bucket
    key = "${var.ssh_key_name}.pem" # Use the same name as the key
(with .pem extension)
```

```
  content                   =
tls_private_key.example_ssh_key.private_key_pem
  acl                       = "private"
  server_side_encryption     = "AES256"
}
```

# Create a Security Group for the web server and SSH access

```
resource "aws_security_group" "web_sg" {
  name        = "web-server-sg"
  vpc_id      = aws_vpc.tp_cloud_devops_vpc.id
  description = "Security group for web server and SSH access"
}
```

# Allow inbound HTTP traffic on port 8080

```
resource "aws_security_group_rule" "allow_web_http_inbound" {
  type              = "ingress"
  security_group_id = aws_security_group.web_sg.id
  from_port         = 8080
  to_port           = 8080
  protocol          = "tcp"
  cidr_blocks       = ["0.0.0.0/0"]
}
```

# Allow inbound HTTP traffic on port 80

```
resource "aws_security_group_rule" "allow_web_http_inbound-80" {
  type              = "ingress"
  security_group_id = aws_security_group.web_sg.id
  from_port         = 80
  to_port           = 80
  protocol          = "tcp"
  cidr_blocks       = ["0.0.0.0/0"]
}
```

# Allow inbound SSH traffic on port 22

```
resource "aws_security_group_rule" "allow_web_ssh_inbound" {
  type              = "ingress"
  security_group_id = aws_security_group.web_sg.id
  from_port         = 22
  to_port           = 22
  protocol          = "tcp"
  cidr_blocks       = ["0.0.0.0/0"]
}
```

# Allow all outbound traffic

```
resource "aws_security_group_rule" "allow_all_outbound" {
  type              = "egress"
  security_group_id = aws_security_group.web_sg.id
  from_port         = 0
  to_port           = 0
  protocol          = "-1"  # "-1" means all protocols
  cidr_blocks       = ["0.0.0.0/0"]
}
```

# Launch an EC2 instance in the public subnet with the generated key pair and security group

```
resource "aws_instance" "public_instance" {
  ami                    = var.ec2_ami_id
  instance_type          = var.instance_type
  subnet_id              = aws_subnet.public_subnet_1.id
  key_name               = aws_key_pair.deployer_key.key_name
  vpc_security_group_ids = [aws_security_group.web_sg.id]
  user_data     = <<-EOF
                 #!/bin/bash
                 echo "<h1>Hello, World</h1>" > index.html
                 # Start a simple HTTP server on port 8080
                 python3 -m http.server 8080 &
                 EOF
  tags = {
    Name = "PublicInstance"
  }
}
```

Updated variables.tf

Now, let's make sure that the variables used in the main.tf are defined properly in the variables.tf file.

# SSH Key Name

```
variable "ssh_key_name" {
  description = "The name of the SSH key pair to be created"
  type        = string
  default     = "deployer_key"
}
# EC2 Instance Type
variable "instance_type" {
  description = "The type of EC2 instance"
  type        = string
  default     = "t2.micro"
}
# EC2 AMI ID
variable "ec2_ami_id" {
  description = "The AMI ID for the EC2 instance"
  type        = string
  default     = "ami-0ebfd941bbafe70c6"
}
```

Create `outputs.tf` to Display Key Information

Once the infrastructure is provisioned, display key outputs such as the public IP address of the EC2 instance.

`outputs.tf`

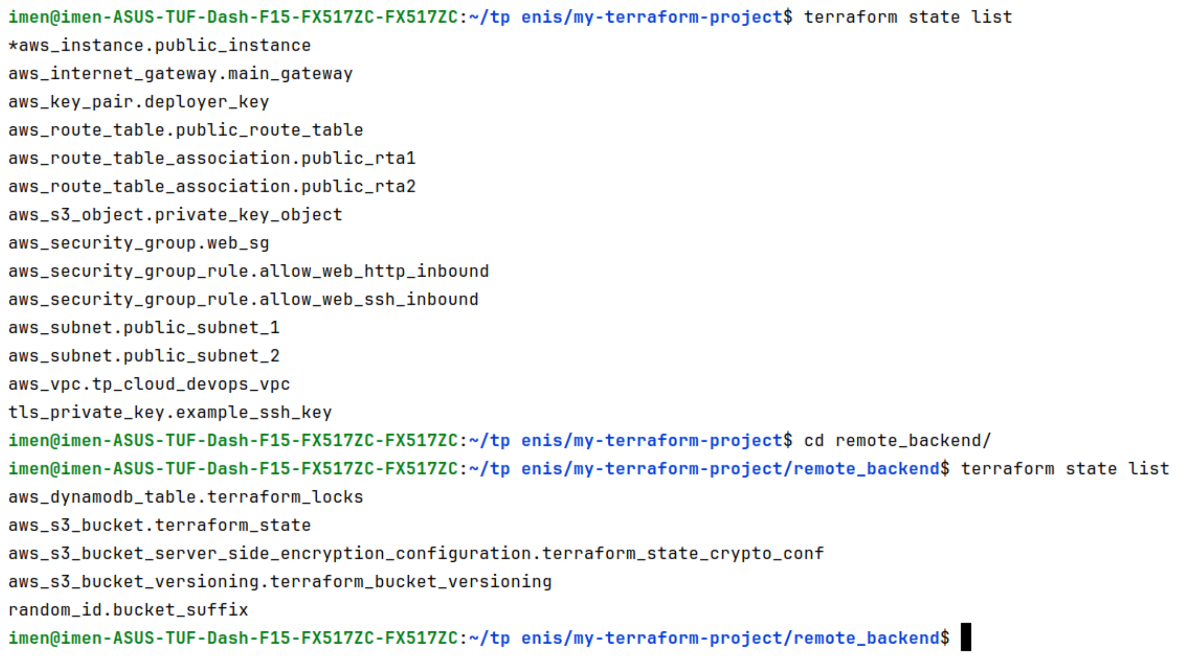
```
# Output all relevant EC2 instance details as a single object
output "instance_details" {
  description = "Detailed information about the EC2 instance"
  value = {
    instance_id          = aws_instance.public_instance.id
    instance_public_ip   = aws_instance.public_instance.public_ip
    instance_private_ip  = aws_instance.public_instance.private_ip
    instance_name        = aws_instance.public_instance.tags["Name"]
    security_group_id = aws_instance.public_instance.vpc_security_group_ids
# Security group attached to the instance
    subnet_id            = aws_instance.public_instance.subnet_id
  }
}
```

Example Output

When you run `terraform apply`, the output will be structured as a single object with all the details:

```
Outputs:

dynamodb_table_name = "custom-terraform-state-locks"
instance_details = {
  "instance_id" = "i-0129b8b6c61a8deb8"
  "instance_name" = "PublicInstance"
  "instance_private_ip" = "10.0.1.51"
  "instance_public_ip" = "54.87.158.108"
  "security_group_ids" = toset([
    "sg-0e4d39d60c38a2c74",
  ])
  "subnet_id" = "subnet-0b1635ccc5dafad65"
}
s3_bucket_arn = "arn:aws:s3:::custom-terraform-state-bucket-e95562ef"
```

You can see the state divided in two folders

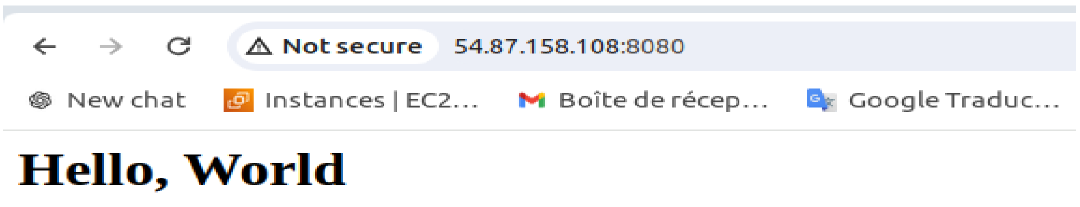


```
imen@imen-ASUS-TUF-Dash-F15-FX517ZC-FX517ZC:~/tp enis/my-terraform-project$ terraform state list
*aws_instance.public_instance
aws_internet_gateway.main_gateway
aws_key_pair.deployer_key
aws_route_table.public_route_table
aws_route_table_association.public_rta1
aws_route_table_association.public_rta2
aws_s3_object.private_key_object
aws_security_group.web_sg
aws_security_group_rule.allow_web_http_inbound
aws_security_group_rule.allow_web_ssh_inbound
aws_subnet.public_subnet_1
aws_subnet.public_subnet_2
aws_vpc.tp_cloud_devops_vpc
tls_private_key.example_ssh_key
imen@imen-ASUS-TUF-Dash-F15-FX517ZC-FX517ZC:~/tp enis/my-terraform-project$ cd remote_backend/
imen@imen-ASUS-TUF-Dash-F15-FX517ZC-FX517ZC:~/tp enis/my-terraform-project/remote_backend$ terraform state list
aws_dynamodb_table.terraform_locks
aws_s3_bucket.terraform_state
aws_s3_bucket_server_side_encryption_configuration.terraform_state_crypto_conf
aws_s3_bucket_versioning.terraform_bucket_versioning
random_id.bucket_suffix
imen@imen-ASUS-TUF-Dash-F15-FX517ZC-FX517ZC:~/tp enis/my-terraform-project/remote_backend$
```

Finally access the EC2 Instance

Once the `apply` command completes, check the output for the Public IP of your instance. You can access the web server via:

type: http://<public-ip-address>:8080



Not secure 54.87.158.108:8080

Hello, World

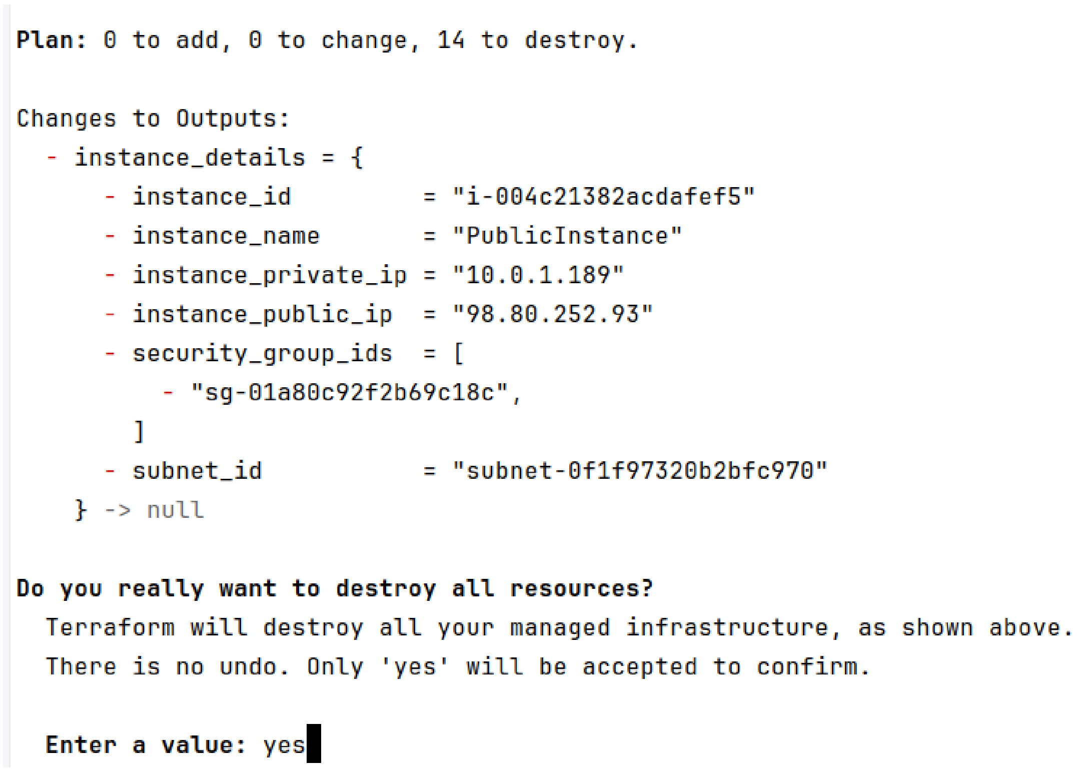
You can also SSH into the instance using the private key stored in S3.

Final Step : Clean Up Resources

To destroy the resources you created:

go to the main folder to destroy the app infrastructure it should be **<u>14 resources</u>**

```
terraform destroy
```



```
Plan: 0 to add, 0 to change, 14 to destroy.

Changes to Outputs:
  - instance_details = {
      - instance_id         = "i-004c21382acdafef5"
      - instance_name       = "PublicInstance"
      - instance_private_ip = "10.0.1.189"
      - instance_public_ip  = "98.80.252.93"
      - security_group_ids  = [
          - "sg-01a80c92f2b69c18c",
        ]
      - subnet_id           = "subnet-0f1f97320b2bfc970"
    } -> null

Do you really want to destroy all resources?
  Terraform will destroy all your managed infrastructure, as shown above.
  There is no undo. Only 'yes' will be accepted to confirm.

  Enter a value: yes
```

After you have finished provisioning resources with Terraform and you decide to **destroy** them, it's important to verify that all resources have been properly removed and that your **Terraform state** is empty.

```
Destroy complete! Resources: 14 destroyed.
```

Here are the final steps to perform a **cleanup check**:

```
terraform state list
```

If all resources have been destroyed, the state list should be **empty**, and you will see the following output:

```
No resources found in the state file
```