Week - 8 Infrastructure As Code - Terraform

- Introduction to Infrastructure as Code (IaC)
- Getting Started with Terraform
- Terraform Basics: Variables, Resources, Attributes, and Dependencies
- Terraform State Management
- Advanced Terraform Concepts: for-each and module
- Terraform Project Development
- AWS Infrastructure Security with Terraform
- CIDR Setup Example with /16
- Subnet Configuration with Terraform
- Terraform State Locking
- Terraform Modules

Introduction to Infrastructure as Code (IaC)

Infrastructure as Code (IaC) is the practice of managing and provisioning computing infrastructure through machine-readable configuration files, rather than through physical hardware configuration or interactive configuration tools. IaC allows for automation, scalability, and consistency in managing infrastructure, making it an essential practice in modern DevOps and cloud computing.

Benefits of IaC:

- Consistency: Ensures that the same environment is created every time.
- Automation: Reduces manual errors and speeds up provisioning.
- Version Control: Infrastructure configurations can be version-controlled just like application code.
- Scalability: Simplifies the process of scaling infrastructure up or down.
- Cost Efficiency: Optimizes resource usage and reduces overhead.

Terraform is an open-source IaC tool developed by HashiCorp that allows you to define and provision infrastructure using a high-level configuration language. Terraform supports a wide range of providers, including AWS, Azure, Google Cloud, and many others.

Basic Workflow:

- Write Configuration Files: Use .tf files to describe the desired state of your infrastructure.
- Initialize Terraform: Run Terraform init to initialize the working directory.
- Plan Infrastructure Changes: Use the Terraform plan to see what changes Terraform will make.

- Apply Changes: Run “terraform apply” to create the infrastructure as defined in your configuration files.
- Destroy Infrastructure: Use terraform destroy to remove the infrastructure.

Terraform Basics: Variables, Resources, Attributes, and Dependencies

Terraform Provider: A provider in Terraform serves as an interface to interact with a specific set of resources. These resources could range from cloud services like AWS, Azure, and Google Cloud Platform.

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "5.47.0"
    }
  }
}

provider "aws" {
  # Configuration options
  region = "ap-south-1"
  profile = "default"
}
```

Variables: Variables in Terraform allow you to parameterize your configurations. You can define variables in variables.tf file and assign values in a terraform.tfvars file or via command-line arguments.

```
variable "region" {
  description = "The AWS region to deploy in"
  default     = "us-west-2"
}
```

Outputs: In Terraform, outputs are a way to extract and display values from your infrastructure configuration. Outputs are useful for displaying important information to the user, for passing data between Terraform configurations, and for integration with other automation tools.

It is going to print the vpc id in the output

```
output "vpc_id" {
  description = "The ID of the VPC"
  value       = aws_vpc.example_vpc.id
}
```

Resources: Resources are the most important element in the Terraform language. Each resource describes one or more infrastructure objects, such as a virtual network or compute instance.

```
resource "aws_instance" "web" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"

  tags = {
    Name = "web-server"
  }
}
```

Attributes: Attributes are the fields of a resource. These can be set explicitly or can be derived from other resources.

```
resource "aws_instance" "web" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"

  tags = {
    Name = "web-server"
  }

  availability_zone = "us-west-2a"
}
```

Dependencies: Dependencies between resources are automatically handled by Terraform. You can also manually specify dependencies using the `depends_on` attribute.

```
resource "aws_instance" "web" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
  depends_on    = [aws_security_group.web_sg]
```

}

Terraform Data Sources:

Data sources in Terraform allow you to query existing infrastructure or services outside of your configuration to obtain information that can be used dynamically within your Terraform code. Unlike resources, which create or manage infrastructure, data sources retrieve information without modifying the state of your resources.

Here is the sample code, It is going to query for the Amazon ami id from AWS Cloud.

```
data "aws_ami" "latest_amazon_linux" {
  most_recent      = true
  owners           = ["amazon"]

  filter {
    name   = "name"
    values = ["al2023-ami-2023.*-x86_64"]
  }

  filter {
    name   = "root-device-type"
    values = ["ebs"]
  }

  filter {
    name   = "virtualization-type"
    values = ["hvm"]
  }
}
```

Terraform State Management

Terraform uses state files to keep track of the infrastructure it manages. This state file is crucial for understanding the current state of resources and planning future changes.

Key Concepts:

- **State File**: A JSON file that maps your configuration to the real-world resources.
- **Remote State**: Store the state file remotely (e.g., in an S3 bucket) for team collaboration.

- **State Locking**: Prevents concurrent state operations, reducing the risk of conflicts and corruption.

Commands:

- `terraform state list`: Lists all resources in the state file.
- `terraform state show <resource>`: Shows detailed information about a resource in the state file.
- `terraform state mv`: Moves an item in the state file.
- `terraform state rm <resource>`: Removes a resource from the state file.

Advanced Terraform Concepts: for-each and Module

for_each: The `for_each` meta-argument allows you to create multiple instances of a resource or module using a map or set.

```
variable "instance_names" {
  type    = set(string)
  default = ["instance1", "instance2"]
}

resource "aws_instance" "example" {
  for_each = var.instance_names

  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
  tags = {
    Name = each.key
  }
}
```

Modules: Modules are a way to encapsulate and reuse configurations. A module is simply a collection of `.tf` files in a directory.
Module Directory Structure:

```
modules/
  webserver/
    main.tf
    variables.tf
    outputs.tf
```

Using a Module:

```
module "webserver" {
  source = "./modules/webserver"
  count  = 3
  ami    = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
}
```

AWS Infrastructure Security with Terraform

Security Best Practices:

1. **IAM Policies**:
    - Use least privilege principles for IAM roles and policies.
    - Define and attach policies to your resources via Terraform.

```
resource "aws_iam_role" "example" {
  name = "example-role"

  assume_role_policy = jsonencode({
    Version = "2012-10-17"
    Statement = [
      {
        Action = "sts:AssumeRole"
        Effect = "Allow"
        Principal = {
          Service = "ec2.amazonaws.com"
        }
      },
    ]
  })
}
```

Encryption:

- Enable encryption for S3 buckets, EBS volumes, and RDS instances.
- Use Terraform to manage encryption settings.

Example:

```
resource "aws_s3_bucket" "example" {
  bucket = "example-bucket"
  server_side_encryption_configuration {
    rule {
      apply_server_side_encryption_by_default {
        sse_algorithm = "AES256"
      }
    }
  }
}
```

Network Security:

- Use security groups and network ACLs to control traffic.
- Implement VPCs, subnets, and routing tables via Terraform.

Example:

```
resource "aws_security_group" "example" {
  name = "example-sg"

  ingress {
    from_port   = 80
    to_port     = 80
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

CIDR Setup Example with /16

CIDR Notation: CIDR (Classless Inter-Domain Routing) is a method for allocating IP addresses and IP routing. A CIDR block is defined by an IP address and a prefix size (e.g., `192.168.0.0/16`).

Example:

- `/16` CIDR block provides 65,536 IP addresses.
- Commonly used for defining a large network range in a VPC.

```
resource "aws_vpc" "example" {
  cidr_block = "192.168.0.0/16"

  tags = {
    Name = "example-vpc"
  }
}
```

Subnet Creation

To make use of the large address space, create subnets within the VPC. This example divides the VPC into two /24 subnets.

- resource "aws_subnet" "example_subnet_1" and resource "aws_subnet" "example_subnet_2": These blocks define two subnet resources.
- vpc_id = aws_vpc.example_vpc.id: Associates the subnets with the VPC created in Step 1.
- cidr_block: Specifies the CIDR block for each subnet (/24 provides 256 IP addresses per subnet).
- availability_zone: Specifies the availability zone for each subnet.
- tags: Adds tags for easier identification.

```
resource "aws_subnet" "example_subnet_1" {
  vpc_id     = aws_vpc.example_vpc.id
  cidr_block = "192.168.1.0/24"
  availability_zone = "us-west-2a"

  tags = {
    Name = "example_subnet_1"
  }
}
```

Internet Gateway

Attach an Internet Gateway to the VPC to allow internet access.

- resource "aws_internet_gateway" "example_igw": Defines an Internet Gateway resource.
- vpc_id = aws_vpc.example_vpc.id: Associates the Internet Gateway with the VPC.
- tags: Adds a tag for easier identification.

```
resource "aws_internet_gateway" "example_igw" {
  vpc_id = aws_vpc.example_vpc.id

  tags = {
    Name = "example-igw"
  }
}
```

Route Table
Create a route table and a default route to the Internet Gateway.

- resource "aws_route_table" "example_route_table": Defines a route table resource.
- vpc_id = aws_vpc.example_vpc.id: Associates the route table with the VPC.
- route: Defines a route within the table.
- cidr_block = "0.0.0.0/0": Specifies that this route applies to all traffic.
- gateway_id = aws_internet_gateway.example_igw.id: Directs the traffic to the Internet Gateway.
- tags: Adds a tag for easier identification.

```
resource "aws_route_table" "example_route_table" {
  vpc_id = aws_vpc.example_vpc.id

  route {
    cidr_block = "0.0.0.0/0"
    gateway_id = aws_internet_gateway.example_igw.id
  }

  tags = {
    Name = "example-route-table"
  }
}
```

Associate Route Table with Subnets

Associate the route table with the subnets to ensure traffic can route through the Internet Gateway.

- resource "aws_route_table_association" "example_route_table_association_1" and resource "aws_route_table_association" "example_route_table_association_2": These blocks associate the route table with each subnet.
- subnet_id: Specifies the subnet to associate.
- route_table_id: Specifies the route table to associate with the subnet.

Association for Subnet:

```
resource "aws_route_table_association" "example_route_table_association_1" {
  subnet_id      = aws_subnet.example_subnet_1.id
  route_table_id = aws_route_table.example_route_table.id
}
```