Day 3: Writing and Managing Terraform Configurations

Terraform Configuration Language (HCL)

Terraform uses its own configuration language called HashiCorp Configuration Language (HCL).

HCL is designed to be both human-readable and machine-friendly, striking a balance between ease of use and powerful features.

Key concepts of HCL:

1. Blocks: Containers for other content that usually represent the configuration of some kind of object, like a resource.

2. Arguments: Assign a value to a name. They appear within blocks.

3. Expressions: Represent a value, either literally or by referencing and combining other values.

Let's look at a basic example of HCL syntax:

```
resource "aws_instance" "example" {

ami = "ami-0c55b159cbfafe1f0"

instance_type = "t2.micro"

 tags = {

Name = "ExampleInstance"

}

}
```

In this example:

- `resource` is a block type

- `"aws_instance"` is the resource type

- `"example"` is the resource name

- The curly braces `{}` contain the configuration for this resource

- `ami`, `instance_type`, and `tags` are arguments

Resource Dependencies and Ordering

Terraform automatically handles resource dependencies based on references within your configuration. For example:

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

resource "aws_eip" "ip" {
instance = aws_instance.web.id
}
```

Terraform understands that the Elastic IP depends on the EC2 instance and will create the instance first.

This diagram illustrates how Terraform orders resource creation based on dependencies. The EC2 instance is created first, as the Elastic IP depends on it. The AMI is shown as a dependency of the EC2 instance.

Advanced Configuration

Using Variables, Outputs, and Data Sources Effectively

Variables allow you to parameterize your configurations, making them more flexible and reusable. Outputs provide a way to expose certain values for use by other Terraform configurations or external scripts. Data sources allow Terraform to use information defined outside of Terraform.

Example showcasing variables, outputs, and data sources:

```
variable "instance_type" {
description = "EC2 instance type"
type = string
default = "t2.micro"
```

```
}
 data "aws_ami" "ubuntu" {
 most_recent = true
 owners = ["099720109477"] # Canonical
  filter {
 name = "name"
 values = ["ubuntu/images/hvm-ssd/ubuntu-focal-20.04-amd64-server-*"]
 }
}
resource "aws_instance" "example" {
 ami = data.aws_ami.ubuntu.id
 instance_type = var.instance_type
  tags = {
 Name = "ExampleInstance"
 }
}
 output "instance_public_ip" {
 description = "Public IP of the EC2 instance"
 value = aws_instance.example.public_ip
}
```

Understanding the Use of Collection and Structural Types

Terraform supports several collection and structural types that allow you to group and structure your data:

1. List: An ordered sequence of values.
2. Map: A collection of key-value pairs.
3. Set: An unordered collection of unique values.

Example using these types:

```
variable "availability_zones" {
```

```
 type = list(string)
 default = ["us-west-2a", "us-west-2b", "us-west-2c"]
}
variable "instance_tags" {
 type = map(string)
 default = {
 Environment = "Production"
 Project = "Terraform"
 }
}
 variable "allowed_ports" {
 type = set(number)
 default = [22, 80, 443]
}
 resource "aws_instance" "example" {
 count = 3
 ami = data.aws_ami.ubuntu.id
 instance_type = var.instance_type
 availability_zone = var.availability_zones[count.index]
 tags = merge(var.instance_tags, {
 Name = "Instance-${count.index + 1}"
 })
}
 resource "aws_security_group" "example" {
 name = "example"
 description = "Allow inbound traffic"
 dynamic "ingress" {
```

```
 for_each = var.allowed_ports

 content {

 from_port = ingress.value

 to_port = ingress.value

 protocol = "tcp"

 cidr_blocks = ["0.0.0.0/0"]

 }

 }

}
```

Industry Insight

Best practices for writing maintainable and scalable Terraform configurations:

1. Use consistent naming conventions for resources, variables, and outputs.

2. Implement proper code organization by separating configurations into logical files and modules.

3. Version control your Terraform configurations using tools like Git.

4. Implement a robust state management strategy, including using remote backends and state locking.

5. Use variables and locals to make your code DRY (Don't Repeat Yourself) and more maintainable.

6. Implement proper error handling and input validation for variables.

7. Use data sources to fetch dynamic information rather than hardcoding values.

8. Implement proper tagging strategies for resources to aid in cost allocation and management.

How companies structure their Terraform codebases:

Many companies adopt a modular approach to structuring their Terraform configurations. This typically involves:

1. Root Module: The main entry point for the Terraform configuration.

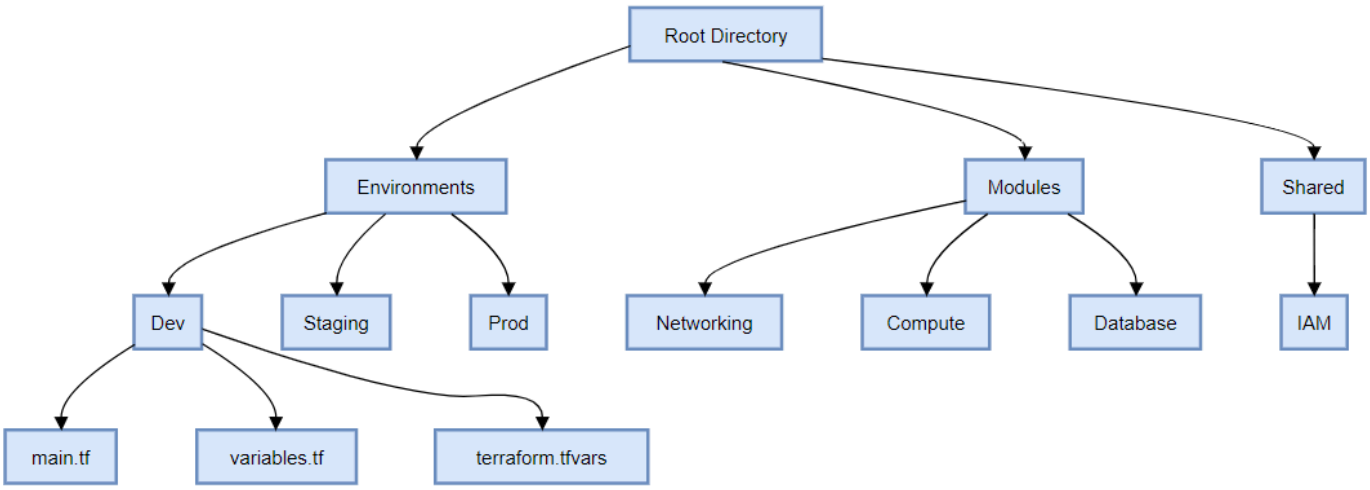
2. Child Modules: Reusable components that define specific resources or sets of resources.

3. Environment-specific configurations: Separate configurations for different environments (dev, staging, prod).

4. Shared Modules: Common modules used across different projects or teams.

Example of how a company might structure their Terraform codebase:

This structure allows for code reuse, separation of concerns, and easier management of different environments.



- This diagram illustrates a typical Terraform project structure, showing the relationships between different directories and files.
- The root directory contains environments, modules, and shared resources.
- Each environment (dev, staging, prod) has its own configuration files.

Hands-On Activity

Let's create a more complex Terraform configuration that sets up a basic web application infrastructure on AWS.

This will include a VPC, subnets, an EC2 instance, and a security group.

```
# Provider configuration

provider "aws" {

 region = var.aws_region

}

 # Variables

variable "aws_region" {

 description = "The AWS region to deploy to"

 type = string

 default = "us-west-2"

}

 variable "vpc_cidr" {

 description = "The CIDR block for the VPC"
```

```
 type = string
 default = "10.0.0.0/16"
}
 variable "subnet_cidrs" {
 description = "The CIDR blocks for the subnets"
 type = list(string)
 default = ["10.0.1.0/24", "10.0.2.0/24"]
}
 variable "instance_type" {
 description = "The EC2 instance type"
 type = string
 default = "t2.micro"
}
 # Data source for latest Amazon Linux 2 AMI
data "aws_ami" "amazon_linux_2" {
 most_recent = true
 owners = ["amazon"]
  filter {
 name = "name"
 values = ["amzn2-ami-hvm-*-x86_64-gp2"]
 }
}
 # VPC
resource "aws_vpc" "main" {
 cidr_block = var.vpc_cidr
 enable_dns_hostnames = true
 tags = {
```

```
 Name = "Main VPC"
 }
}
# Internet Gateway
resource "aws_internet_gateway" "main" {
 vpc_id = aws_vpc.main.id
 tags = {
 Name = "Main IGW"
 }
}
 # Subnets
resource "aws_subnet" "public" {
 count = length(var.subnet_cidrs)
 vpc_id = aws_vpc.main.id
 cidr_block = var.subnet_cidrs[count.index]
 availability_zone = data.aws_availability_zones.available.names[count.index]
 map_public_ip_on_launch = true
 tags = {
 Name = "Public Subnet ${count.index + 1}"
 }
}
 # Route Table
resource "aws_route_table" "public" {
 vpc_id = aws_vpc.main.id

 route {
 cidr_block = "0.0.0.0/0"
```

```
 gateway_id = aws_internet_gateway.main.id
 }
  tags = {
 Name = "Public Route Table"
 }
}
 # Route Table Association
resource "aws_route_table_association" "public" {
 count = length(aws_subnet.public)
 subnet_id = aws_subnet.public[count.index].id
 route_table_id = aws_route_table.public.id
}
 # Security Group
resource "aws_security_group" "web" {
 name = "allow_web"
 description = "Allow inbound web traffic"
 vpc_id = aws_vpc.main.id
  ingress {
 description = "HTTP from anywhere"
 from_port = 80
 to_port = 80
 protocol = "tcp"
 cidr_blocks = ["0.0.0.0/0"]
 }
  ingress {
 description = "SSH from anywhere"
 from_port = 22
```

```
 to_port = 22
 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"]
 }
tags = {
 Name = "Allow Web Traffic"
 }
}
 # EC2 Instance
resource "aws_instance" "web" {
 ami = data.aws_ami.amazon_linux_2.id
 instance_type = var.instance_type
 subnet_id = aws_subnet.public[0].id
  vpc_security_group_ids = [aws_security_group.web.id]
 user_data = <<-EOF
 #!/bin/bash
 yum update -y
 yum install -y httpd
 systemctl start httpd
 systemctl enable httpd
 echo "<h1>Hello from Terraform</h1>" > /var/www/html/index.html
```

```
 EOF
  tags = {
 Name = "Web Server"
 }
}
 # Outputs
output "vpc_id" {
 description = "ID of the created VPC"
 value = aws_vpc.main.id
}
 output "public_subnet_ids" {
 description = "IDs of the created public subnets"
 value = aws_subnet.public[*].id
}
 output "web_instance_public_ip" {
 description = "Public IP of the web server"
 value = aws_instance.web.public_ip
}
```

To use this configuration:

1. Save the content in a file named `main.tf`.

2. Initialize the Terraform working directory:

terraform init

3. Review the planned changes:

terraform plan

4. Apply the configuration:

terraform apply

This configuration will create a VPC with two public subnets, an Internet Gateway, a route table, a security group, and an EC2 instance running a simple web server.

Real-world Scenarios

1. Multi-environment Deployments:

- Companies often need to manage multiple environments (dev, staging, prod) with similar but slightly different configurations.
- They might use workspace-specific variables or separate configuration files for each environment.

2. Compliance and Security:

- Organizations in regulated industries need to ensure their infrastructure meets specific compliance requirements.
- They might use Terraform to enforce security group rules, encrypt data at rest, and implement proper IAM policies.

3. Disaster Recovery:

Companies implement disaster recovery strategies using Terraform by creating redundant infrastructure in different regions or availability zones.

4. Microservices Architecture:

Terraform is used to provision and manage the infrastructure for microservices, including container orchestration platforms like Kubernetes.

5. Cost Optimization:

Organizations use Terraform to implement auto-scaling groups, scheduled scaling actions, and resource tagging to optimize their cloud spend.

1. What is the purpose of the `count` meta-argument in Terraform?

a) To count the number of resources

b) To create multiple instances of a resource

c) To limit the number of API calls

d) To set a timeout for resource creation

2. Which of the following is NOT a valid collection type in Terraform?

a) List

b) Map

c) Set

d) Array

3. How does Terraform determine the order in which to create resources?

a) Alphabetically by resource name

b) In the order they appear in the configuration file

c) Based on implicit and explicit dependencies

d) Randomly

4. What is the purpose of the `terraform.tfvars` file?

a) To define provider configurations

b) To specify variable values

c) To store the Terraform state

d) To define output values

5. Which of the following is a best practice for organizing Terraform code?

a) Putting all resources in a single file

b) Using modules for reusable components

c) Hardcoding all values in the configuration

d) Avoiding the use of variables

Answers to Exam Practice Questions:

1. b) To create multiple instances of a resource

2. d) Array (Terraform supports List, Map, and Set, but not Array)

3. c) Based on implicit and explicit dependencies

4. b) To specify variable values

5. b) Using modules for reusable components

Let's add some explanations for these answers:

1. The `count` meta-argument is used to create multiple instances of a resource.

It's particularly useful when you need to create a number of similar resources, like multiple EC2 instances or subnets.

2. Terraform supports three collection types: List (ordered collection), Map (collection of key-value pairs), and Set (unordered collection of unique values).

Array is not a valid collection type in Terraform.

3. Terraform creates a dependency graph based on the references in your configuration.

It uses this graph to determine the order in which resources should be created, updated, or deleted.

4. The `terraform.tfvars` file is used to set values for variables defined in your Terraform configuration.

This allows you to separate your variable definitions from their values, making it easier to manage different environments.

5. Using modules for reusable components is a best practice in Terraform.

It promotes code reuse, makes your configurations more maintainable, and allows you to encapsulate complex logic.

Additional Content: Terraform Modules

- Terraform modules are containers for multiple resources that are used together.
- They allow you to create reusable components, improve organization, and encapsulate groups of resources dedicated to completing a specific task.

Example of how you might structure a module:

```
# modules/ec2-instance/main.tf

variable "instance_type" {

description = "The type of instance to start"

type = string

default = "t2.micro"

}

variable "ami_id" {

description = "The AMI to use for the instance"

type = string

}

variable "subnet_id" {

description = "The VPC Subnet ID to launch in"

type = string

}

variable "tags" {

description = "A mapping of tags to assign to the resource"

type = map(string)

default = {}

}

resource "aws_instance" "this" {

ami = var.ami_id
```

```
 instance_type = var.instance_type
 subnet_id = var.subnet_id
  tags = var.tags
}
 output "instance_id" {
 description = "The ID of the instance"
 value = aws_instance.this.id
}
 output "private_ip" {
 description = "The private IP address assigned to the instance"
 value = aws_instance.this.private_ip
}
# main.tf (in the root module)
 module "ec2_instance" {
 source = "./modules/ec2-instance"
 instance_type = "t2.micro"
 ami_id = "ami-0c55b159cbfafe1f0"
 subnet_id = aws_subnet.example.id
 tags = {
 Name = "ExampleInstance"
 }
}
 output "instance_id" {
 value = module.ec2_instance.instance_id
}
```

In this example, we've created a module for an EC2 instance.

The module is defined in its own directory (`modules/ec2-instance`) and includes variables for customization, the resource definition, and outputs.

In the root module (`main.tf`), we can then use this module by referencing it and providing the required inputs.

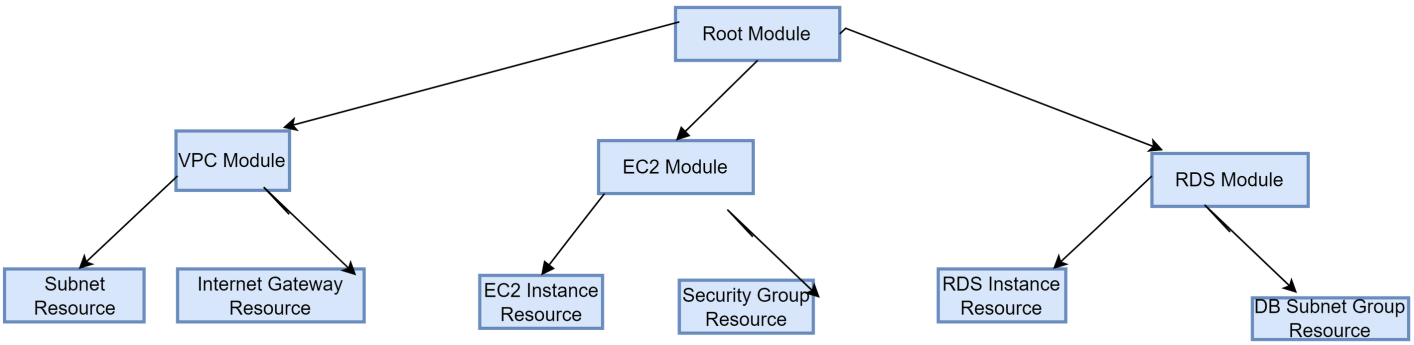
Using modules offers several benefits:

1. **Encapsulation**: Modules allow you to group related resources together, making your configurations easier to understand and maintain.

2. **Reusability**: You can use the same module in multiple places in your configuration or even across different projects.

3. **Abstraction**: Modules can hide complex implementation details behind a simpler interface.

4. **Consistency**: By using modules, you can ensure that resources are created consistently across your infrastructure.



This diagram shows how a root module can use multiple child modules (VPC, EC2, RDS), each of which encapsulates related resources.

This structure promotes reusability and maintainability in your Terraform configurations.

Conclusion

- Day 3 has covered advanced aspects of writing and managing Terraform configurations.
- We've explored the Terraform Configuration Language (HCL) in depth, including resource dependencies, advanced configuration techniques, and the use of variables, outputs, and data sources.
- We've also looked at industry best practices for structuring Terraform code and introduced the concept of modules for creating reusable components.
- Remember to apply these concepts in your real-world projects, always considering maintainability, scalability, and security in your infrastructure-as-code practices.
- As you continue your Terraform journey, you'll find that these advanced techniques will help you create more robust and flexible infrastructure deployments.