Infrastructure as Code(IaC)

Before the advent of IaC, infrastructure management was typically a manual and time-consuming process. System administrators and operations teams had to:

1. Manually Configure Servers: Servers and other infrastructure components were often set up and configured manually, which could lead to inconsistencies and errors.

2. Lack of Version Control: Infrastructure configurations were not typically version-controlled, making it difficult to track changes or revert to previous states.

3. Documentation Heavy: Organizations relied heavily on documentation to record the steps and configurations required for different infrastructure setups. This documentation could become outdated quickly.

4. Limited Automation: Automation was limited to basic scripting, often lacking the robustness and flexibility offered by modern IaC tools.

5. Slow Provisioning: Provisioning new resources or environments was a time-consuming process that involved multiple manual steps, leading to delays in project delivery.

IaC addresses these challenges by providing a systematic, automated, and code-driven approach to infrastructure management. Popular IaC tools include Terraform, AWS CloudFormation, Azure Resource Manager templates others.

These tools enable organizations to define, deploy, and manage their infrastructure efficiently and consistently, making it easier to adapt to the dynamic needs of modern applications and services.

Why Terraform ?

There are multiple reasons why Terraform is used over the other IaC tools but below are the main reasons.

1. **Multi-Cloud Support**: Terraform is known for its multi-cloud support. It allows you to define infrastructure in a cloud-agnostic way, meaning you can use the same configuration code to provision resources on various cloud providers (AWS, Azure, Google Cloud, etc.) and even on-premises infrastructure. This flexibility can be beneficial if your organization uses multiple cloud providers or plans to migrate between them.

2. **Large Ecosystem**: Terraform has a vast ecosystem of providers and modules contributed by both HashiCorp (the company behind Terraform) and the community. This means you can find pre-built modules and configurations for a wide range of services and infrastructure components, saving you time and effort in writing custom configurations.

3. **Declarative Syntax**: Terraform uses a declarative syntax, allowing you to specify the desired end-state of your infrastructure. This makes it easier to understand and maintain your code compared to imperative scripting languages.

4. **State Management**: Terraform maintains a state file that tracks the current state of your infrastructure. This state file helps Terraform understand the

differences between the desired and actual states of your infrastructure, enabling it to make informed decisions when you apply changes.

5. **Plan and Apply**: Terraform's "plan" and "apply" workflow allows you to preview changes before applying them. This helps prevent unexpected modifications to your infrastructure and provides an opportunity to review and approve changes before they are implemented.

6. **Community Support**: Terraform has a large and active user community, which means you can find answers to common questions, troubleshooting tips, and a wealth of documentation and tutorials online.

7. **Integration with Other Tools**: Terraform can be integrated with other DevOps and automation tools, such as Docker, Kubernetes, Ansible, and Jenkins, allowing you to create comprehensive automation pipelines.

8. **HCL Language**: Terraform uses HashiCorp Configuration Language (HCL), which is designed specifically for defining infrastructure. It's human-readable and expressive, making it easier for both developers and operators to work with.

Providers

A provider in Terraform is a plugin that enables interaction with an API. This includes cloud providers, SaaS providers, and other APIs. The providers are specified in the Terraform configuration code. They tell Terraform which services it needs to interact with.

For example, if you want to use Terraform to create a virtual machine on AWS, you would need to use the aws provider. The aws provider provides a set of resources that Terraform can use to create, manage, and destroy virtual machines on AWS.

Here is an example of how to use the aws provider in a Terraform configuration:

```
provider "aws" {
  region = "us-east-1"
}

resource "aws_instance" "example" {
  ami = "ami-0123456789abcdef0" # Change the AMI
  instance_type = "t2.micro"
}
```

In this example, we are first defining the aws provider. We are specifying the region as us-east-1. Then, we are defining the `aws_instance` resource. We are specifying the `AMI ID` and the `instance type`.
When Terraform runs, it will first install the aws provider. Then, it will use the aws provider to create the virtual machine.

Here are some other examples of providers:

- `azurerm` - for Azure
- `google` - for Google Cloud Platform
- `kubernetes` - for Kubernetes
- `openstack` - for OpenStack
- `vsphere` - for VMware vSphere

There are many other providers available, and new ones are being added all the time.

Providers are an essential part of Terraform. They allow Terraform to interact with a wide variety of cloud providers and other APIs. This makes Terraform a very versatile tool that can be used to manage a wide variety of infrastructure.

Different ways to configure providers in terraform

There are three main ways to configure providers in Terraform:

In the root module

This is the most common way to configure providers. The provider configuration block is placed in the root module of the Terraform configuration. This makes the provider configuration available to all the resources in the configuration.

```
provider "aws" {
  region = "us-east-1"
}

resource "aws_instance" "example" {
  ami = "ami-0123456789abcdef0"
  instance_type = "t2.micro"
}
```

In a child module

You can also configure providers in a child module. This is useful if you want to reuse the same provider configuration in multiple resources.

```
module "aws_vpc" {
  source = "./aws_vpc"
  providers = {
    aws = aws.us-west-2
  }
}

resource "aws_instance" "example" {
  ami = "ami-0123456789abcdef0"
  instance_type = "t2.micro"
  depends_on = [module.aws_vpc]
}
```

In the required_providers block

You can also configure providers in the required_providers block. This is useful if you want to make sure that a specific provider version is used.

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "~> 3.79"
    }
  }
}

resource "aws_instance" "example" {
  ami = "ami-0123456789abcdef0"
  instance_type = "t2.micro"
}
```

The best way to configure providers depends on your specific needs. If you are only using a single provider, then configuring it in the root module is the simplest option. If

you are using multiple providers, or if you want to reuse the same provider configuration in multiple resources, then configuring it in a child module is a good option. And if you want to make sure that a specific provider version is used, then configuring it in the required_providers block is the best option.

Multiple Providers

You can use multiple providers in one single terraform project. For example,

1. Create a providers.tf file in the root directory of your Terraform project.
2. In the providers.tf file, define the AWS and Azure providers. For example:

```
provider "aws" {
  region = "us-east-1"
}

provider "azurerm" {
  subscription_id = "your-azure-subscription-id"
  client_id = "your-azure-client-id"
  client_secret = "your-azure-client-secret"
  tenant_id = "your-azure-tenant-id"
}
```

3. In your other Terraform configuration files, you can then use the aws and azurerm providers to create resources in AWS and Azure, respectively,

```
resource "aws_instance" "example" {
  ami = "ami-0123456789abcdef0"
  instance_type = "t2.micro"
}

resource "azurerm_virtual_machine" "example" {
  name = "example-vm"
  location = "eastus"
  size = "Standard_A1"
}
```

Multiple Providers

You can use multiple providers in one single terraform project. For example,

1. Create a providers.tf file in the root directory of your Terraform project.
2. In the providers.tf file, define the AWS and Azure providers. For example:

```
provider "aws" {
  region = "us-east-1"
}

provider "azurerm" {
  subscription_id = "your-azure-subscription-id"
  client_id = "your-azure-client-id"
  client_secret = "your-azure-client-secret"
  tenant_id = "your-azure-tenant-id"
}
```

3. In your other Terraform configuration files, you can then use the aws and azurerm providers to create resources in AWS and Azure, respectively,

```
resource "aws_instance" "example" {
  ami = "ami-0123456789abcdef0"
  instance_type = "t2.micro"
```

```
}

resource "azurerm_virtual_machine" "example" {
  name = "example-vm"
  location = "eastus"
  size = "Standard_A1"
}
```

Multiple Providers

You can use multiple providers in one single terraform project. For example,

1. Create a providers.tf file in the root directory of your Terraform project.
2. In the providers.tf file, define the AWS and Azure providers. For example:

```
provider "aws" {
  region = "us-east-1"
}

provider "azurerm" {
  subscription_id = "your-azure-subscription-id"
  client_id = "your-azure-client-id"
  client_secret = "your-azure-client-secret"
  tenant_id = "your-azure-tenant-id"
}
```

3. In your other Terraform configuration files, you can then use the aws and azurerm providers to create resources in AWS and Azure, respectively,

```
resource "aws_instance" "example" {
  ami = "ami-0123456789abcdef0"
  instance_type = "t2.micro"
}

resource "azurerm_virtual_machine" "example" {
  name = "example-vm"
  location = "eastus"
  size = "Standard_A1"
}
```

Variables

Input and output variables in Terraform are essential for parameterizing and sharing values within your Terraform configurations and modules. They allow you to make your configurations more dynamic, reusable, and flexible.

Input Variables

Input variables are used to parameterize your Terraform configurations. They allow you to pass values into your modules or configurations from the outside. Input variables can be defined within a module or at the root level of your configuration. Here's how you define an input variable:

```
variable "example_var" {
  description = "An example input variable"
  type        = string
  default     = "default_value"
}
```

In this example:

- `variable` is used to declare an input variable named `example_var`.

- `description` provides a human-readable description of the variable.
- `type` specifies the data type of the variable (e.g., `string`, `number`, `list`, `map`, etc.).
- `default` provides a default value for the variable, which is optional.

You can then use the input variable within your module or configuration like this:

```
resource "example_resource" "example" {
  name = var.example_var
  # other resource configurations
}
```

You reference the input variable using `var.example_var`.

Output Variables

Output variables allow you to expose values from your module or configuration, making them available for use in other parts of your Terraform setup. Here's how you define an output variable:

```
output "example_output" {
  description = "An example output variable"
  value       = resource.example_resource.example.id
}
```

In this example:

- `output` is used to declare an output variable named `example_output`.
- `description` provides a description of the output variable.
- `value` specifies the value that you want to expose as an output variable. This value can be a resource attribute, a computed value, or any other expression.

You can reference output variables in the root module or in other modules by using the syntax `module.module_name.output_name`, where `module_name` is the name of the module containing the output variable.
For example, if you have an output variable named `example_output` in a module called `example_module`, you can access it in the root module like this:

```
output "root_output" {
  value = module.example_module.example_output
}
```

This allows you to share data and values between different parts of your Terraform configuration and create more modular and maintainable infrastructure-as-code setups.

Variables Demo

```hcl

# Define an input variable for the EC2 instance type

variable "instance_type" {

 description = "EC2 instance type"

 type       = string

 default    = "t2.micro"

}

# Define an input variable for the EC2 instance AMI ID
```

```
variable "ami_id" {
  description = "EC2 AMI ID"
  type        = string
}

# Configure the AWS provider using the input variables
provider "aws" {
  region     = "us-east-1"
}

# Create an EC2 instance using the input variables
resource "aws_instance" "example_instance" {
  ami           = var.ami_id
  instance_type = var.instance_type
}

# Define an output variable to expose the public IP address of the EC2 instance
output "public_ip" {
  description = "Public IP address of the EC2 instance"
  value       = aws_instance.example_instance.public_ip
}
```

```Terraform tfvars

In Terraform, `.tfvars` files (typically with a `.tfvars` extension) are used to set specific values for input variables defined in your Terraform configuration.
They allow you to separate configuration values from your Terraform code, making it easier to manage different configurations for different environments (e.g., development, staging, production) or to store sensitive information without exposing it in your code.

Here's the purpose of `.tfvars` files:

1. **Separation of Configuration from Code**: Input variables in Terraform are meant to be configurable so that you can use the same code with different sets of values. Instead of hardcoding these values directly into your `.tf` files, you use `.tfvars` files to keep the configuration separate. This makes it easier to maintain and manage configurations for different environments.
2. **Sensitive Information**: `.tfvars` files are a common place to store sensitive information like API keys, access credentials, or secrets. These sensitive values can be kept outside the version control system, enhancing security and preventing accidental exposure of secrets in your codebase.
3. **Reusability**: By keeping configuration values in separate `.tfvars` files, you can reuse the same Terraform code with different sets of variables. This is useful for

creating infrastructure for different projects or environments using a single set of Terraform modules.
4. **Collaboration**: When working in a team, each team member can have their own `.tfvars` file to set values specific to their environment or workflow. This avoids conflicts in the codebase when multiple people are working on the same Terraform project.

Summary

Here's how you typically use `.tfvars` files
1. Define your input variables in your Terraform code (e.g., in a `variables.tf` file).
2. Create one or more `.tfvars` files, each containing specific values for those input variables.

3. When running Terraform commands (e.g., terraform apply, terraform plan), you can specify which .tfvars file(s) to use with the -var-file option:

```
terraform apply -var-file=dev.tfvars
```
By using `.tfvars` files, you can keep your Terraform code more generic and flexible while tailoring configurations to different scenarios and environments.

Conditional Expressions

Conditional expressions in Terraform are used to define conditional logic within your configurations. They allow you to make decisions or set values based on conditions. Conditional expressions are typically used to control whether resources are created or configured based on the evaluation of a condition.

The syntax for a conditional expression in Terraform is:

```
condition ? true_val : false_val
```

- `condition` is an expression that evaluates to either `true` or `false`.
- `true_val` is the value that is returned if the condition is `true`.
- `false_val` is the value that is returned if the condition is `false`.

Here are some common use cases and examples of how to use conditional expressions in Terraform:

Conditional Resource Creation Example

```
resource "aws_instance" "example" {
  count = var.create_instance ? 1 : 0

  ami           = "ami-XXXXXXXXXXXXXXXXX"
  instance_type = "t2.micro"
}
```
In this example, the `count` attribute of the `aws_instance` resource uses a conditional expression. If the `create_instance` variable is `true`, it creates one EC2 instance. If `create_instance` is `false`, it creates zero instances, effectively skipping resource creation.

Conditional Variable Assignment Example

```
variable "environment" {
  description = "Environment type"
```

```
  type        = string
  default     = "development"
}

variable "production_subnet_cidr" {
  description = "CIDR block for production subnet"
  type        = string
  default     = "10.0.1.0/24"
}

variable "development_subnet_cidr" {
  description = "CIDR block for development subnet"
  type        = string
  default     = "10.0.2.0/24"
}

resource "aws_security_group" "example" {
  name        = "example-sg"
  description = "Example security group"

  ingress {
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = var.environment == "production" ? [var.production_subnet_cidr] :
[var.development_subnet_cidr]
  }
}
```

In this example, the `locals` block uses a conditional expression to assign a value to the `subnet_cidr` local variable based on the value of the `environment` variable. If `environment` is set to `"production"`, it uses the `production_subnet_cidr` variable; otherwise, it uses the `development_subnet_cidr` variable.

Conditional Resource Configuration

```
resource "aws_security_group" "example" {
  name = "example-sg"
  description = "Example security group"

  ingress {
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = var.enable_ssh ? ["0.0.0.0/0"] : []
  }
}
```

In this example, the `ingress` block within the `aws_security_group` resource uses a conditional expression to control whether SSH access is allowed. If `enable_ssh` is `true`, it allows SSH traffic from any source (`"0.0.0.0/0"`); otherwise, it allows no inbound traffic. Conditional expressions in Terraform provide a powerful way to make decisions and customize your infrastructure deployments based on various conditions and variables. They enhance the flexibility and reusability of your Terraform configurations.

Built-in Functions

Terraform is an infrastructure as code (IaC) tool that allows you to define and provision infrastructure resources in a declarative manner. Terraform provides a wide range of built-in functions that you can use within your configuration files (usually written in HashiCorp Configuration Language, or HCL) to manipulate and transform data. These functions help you perform various tasks when defining your infrastructure. Here are some commonly used built-in functions in Terraform:

1. `concat(list1, list2, ...)`: Combines multiple lists into a single list.

```
variable "list1" {
  type    = list
  default = ["a", "b"]
}

variable "list2" {
  type    = list
  default = ["c", "d"]
}

output "combined_list" {
  value = concat(var.list1, var.list2)
}
```

2. `element(list, index)`: Returns the element at the specified index in a list.

```
variable "my_list" {
  type    = list
  default = ["apple", "banana", "cherry"]
}

output "selected_element" {
  value = element(var.my_list, 1) # Returns "banana"
}
```

3. `length(list)`: Returns the number of elements in a list.

```
variable "my_list" {
  type    = list
  default = ["apple", "banana", "cherry"]
}

output "list_length" {
  value = length(var.my_list) # Returns 3
}
```

4. `map(key, value)`: Creates a map from a list of keys and a list of values.

```
variable "keys" {
  type    = list
  default = ["name", "age"]
}

variable "values" {
  type    = list
  default = ["Alice", 30]
}

output "my_map" {
  value = map(var.keys, var.values) # Returns {"name" = "Alice", "age" = 30}
}
```

5. `lookup(map, key)`: Retrieves the value associated with a specific key in a map.

```
variable "my_map" {
  type    = map(string)
  default = {"name" = "Alice", "age" = "30"}
}

output "value" {
  value = lookup(var.my_map, "name") # Returns "Alice"
}
```

6. `join(separator, list)`: Joins the elements of a list into a single string using the specified separator.

```
variable "my_list" {
  type    = list
  default = ["apple", "banana", "cherry"]
}

output "joined_string" {
  value = join(", ", var.my_list) # Returns "apple, banana, cherry"
}
```

These are just a few examples of the built-in functions available in Terraform. You can find more functions and detailed documentation in the official Terraform documentation, which is regularly updated to include new features and improvements Certainly, let's delve deeper into the `file`, `remote-exec`, and `local-exec` provisioners in Terraform, along with examples for each.

1. **file Provisioner:**

   The `file` provisioner is used to copy files or directories from the local machine to a remote machine. This is useful for deploying configuration files, scripts, or other assets to a provisioned instance.

   Example:

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

   provisioner "file" {
     source      = "local/path/to/localfile.txt"
     destination = "/path/on/remote/instance/file.txt"
     connection {
       type     = "ssh"
       user     = "ec2-user"
       private_key = file("~/.ssh/id_rsa")
     }
   }
   ```

   In this example, the `file` provisioner copies the `localfile.txt` from the local machine to the `/path/on/remote/instance/file.txt` location on the AWS EC2 instance using an SSH connection.

2. **remote-exec Provisioner:**

   The `remote-exec` provisioner is used to run scripts or commands on a remote machine over SSH or WinRM connections. It's often used to configure or install software on provisioned instances.

   Example:

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

   provisioner "remote-exec" {
     inline = [
       "sudo yum update -y",
       "sudo yum install -y httpd",
       "sudo systemctl start httpd",
     ]

     connection {
       type        = "ssh"
       user        = "ec2-user"
       private_key = file("~/.ssh/id_rsa")
   ```

```
    host        = aws_instance.example.public_ip
  }
}
```

In this example, the `remote-exec` provisioner connects to the AWS EC2 instance using SSH and runs a series of commands to update the package repositories, install Apache HTTP Server, and start the HTTP server.

3. **local-exec Provisioner:**

   The `local-exec` provisioner is used to run scripts or commands locally on the machine where Terraform is executed. It is useful for tasks that don't require remote execution, such as initializing a local database or configuring local resources.

   Example:

```
resource "null_resource" "example" {
  triggers = {
    always_run = "${timestamp()}"
  }

  provisioner "local-exec" {
    command = "echo 'This is a local command'"
  }
}
```

   In this example, a `null_resource` is used with a `local-exec` provisioner to run a simple local command that echoes a message to the console whenever Terraform is applied or refreshed. The `timestamp()` function ensures it runs each time.

Ways to secure Terraform

There are a few ways to manage sensitive information in Terraform files. Here are some of the most common methods:

Use the sensitive attribute

- Terraform provides a sensitive attribute that can be used to mark variables and outputs as sensitive. When a variable or output is marked as sensitive, Terraform will not print its value in the console output or in the state file.

Secret management system

- Store sensitive data in a secret management system. A secret management system is a dedicated system for storing sensitive data, such as passwords, API keys, and SSH keys. Terraform can be configured to read secrets from a secret management system, such as HashiCorp Vault or AWS Secrets Manager.

Remote Backend

- Encrypt sensitive data. The Terraform state file can be encrypted to protect sensitive data. This can be done by using a secure remote backend, such as Terraform Cloud or S3.

Environment Variables

- Use environment variables. Sensitive data can also be stored in environment variables. Terraform can read environment variables when it is run.

Here are some specific examples of how to use these methods:

To mark a variable as sensitive, you would add the sensitive attribute to the variable declaration.

For example:

variable "aws_access_key_id" { sensitive = true }

To store sensitive data in a secret management system, you would first create a secret in the secret management system. Then, you would configure Terraform to read the secret from the secret management system.

For example, to read a secret from HashiCorp Vault, you would use the vault_generic_secret data source.

data "vault_generic_secret" "aws_access_key_id" { path = "secret/aws/access_key_id" }

variable "aws_access_key_id" { value = data.vault_generic_secret.aws_access_key_id.value }

To encrypt the Terraform state file, you would first configure a secure remote backend for the state file. Then, you would encrypt the state file using the terraform encrypt command.

terraform encrypt

To use environment variables, you would first define the environment variables in your operating system. Then, you would configure Terraform to read the environment variables when it is run.

For example, to define an environment variable called AWS_ACCESS_KEY_ID, you would use the following command:

export AWS_ACCESS_KEY_ID=YOUR_ACCESS_KEY_ID

Then, you would configure Terraform to read the environment variable by adding the following line to your Terraform configuration file:

variable "aws_access_key_id" { source = "env://AWS_ACCESS_KEY_ID" }