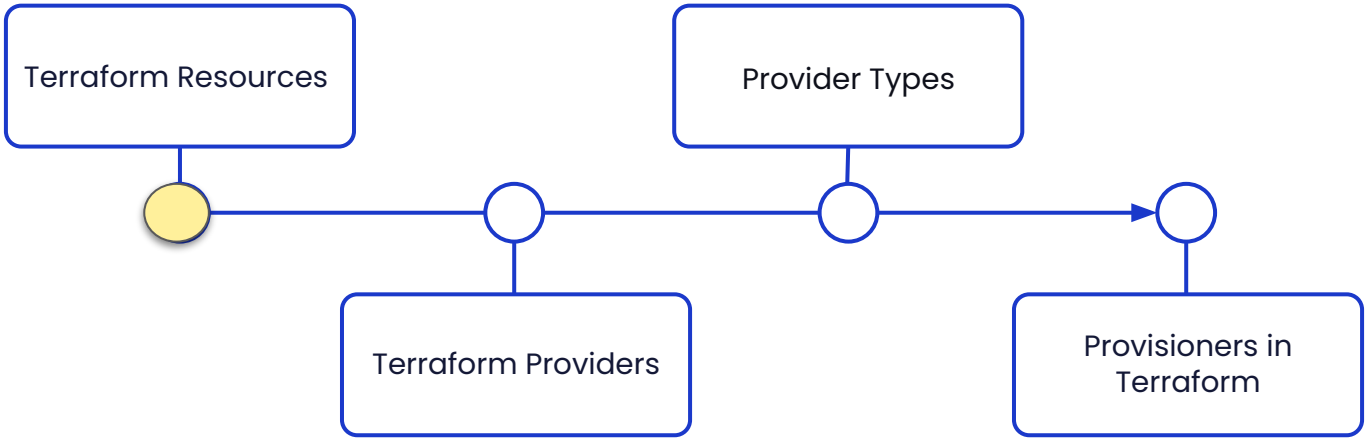
Digital Career Institute

AWS SSA

Terraform | Week 1

DAY : 2
Resources and Providers

Terraform


Terraform Resources
Provider Types
Terraform Providers
Provisioners in Terraform

What Are Terraform Resources?

- The fundamental building blocks
- Managed infrastructure objects
- Declared using the resource block
- Each resource belongs to a specific provider and is declared using the resource block syntax

```
resource "aws_instance" "example"
{
  ami           = "ami-123456"
  instance_type = "t2.micro"
}
```

Resource Configuration Syntax

- Resource blocks contain arguments
- Arguments vary by provider
- Support for both required and optional parameters
- Optional arguments allow for further customization of resources, such as tags or user_data. For example:

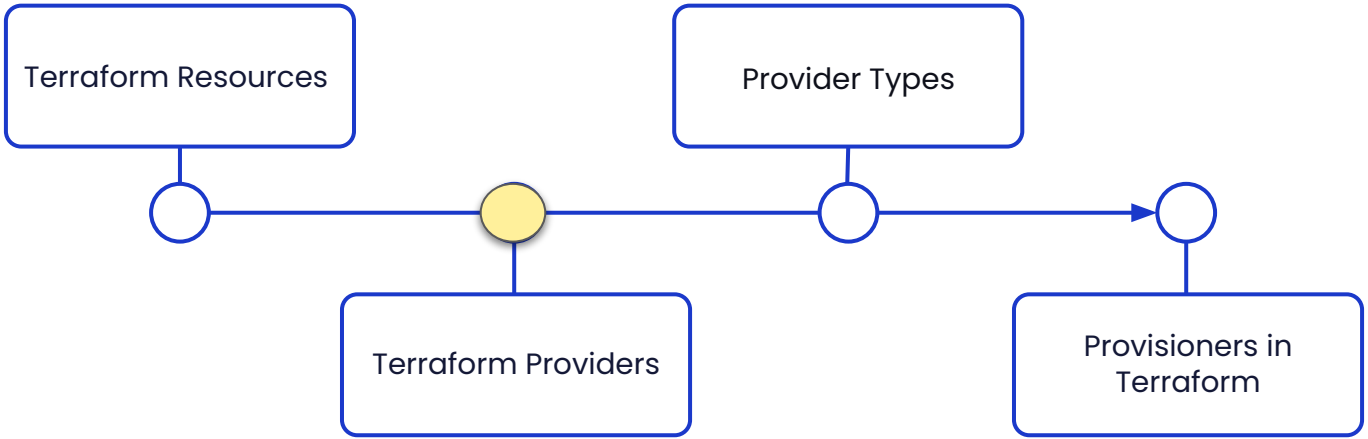
```
resource "aws_instance" "example"
{
  ami           = "ami-123456"
  instance_type = "t2.micro"
  tags = {
    Name = "MyInstance"
  }
}
```

Terraform Resource Lifecycle Management

- Control resource lifecycle: create, read, update, delete
- Customize lifecycle using lifecycle block
- Examples: prevent destroy, ignore changes

```
resource "aws_instance" "example"
{
  ami           = "ami-123456"
  instance_type = "t2.micro"

  lifecycle {
    prevent_destroy = true
    ignore_changes  = [ami]
  }
}
```

Terraform


Terraform Resources
Provider Types
Terraform Providers
Provisioners in Terraform

What Are Terraform Providers

- Connect Terraform to external APIs
- Used to interact with cloud providers (AWS, Azure, GCP)
- Hundreds of providers available

Provider Configuration

- Define providers in your .tf files
- Provider configuration syntax
- Specify versions and credentials

```
provider "aws" {
  region = "us-west-2"
  version = "~> 3.0"
}
```

Providers and Resources: Working Together

- Providers define the APIs Terraform can interact with
- Resources define what will be created, modified, or destroyed
- Multiple providers can be used in a single configuration
- You can define resources from multiple providers in a single configuration file.
- For example, you could manage resources from AWS and Google Cloud within the same Terraform configuration

```
provider "aws" {
  region = "us-west-2"
}

provider "google" {
  project = "my-project"
  region  = "us-central1"
}

resource "aws_instance" "example"
{
  ami           = "ami-123456"
  instance_type = "t2.micro"
}

resource
"google_compute_instance"
"example" {
  name         =
"example-instance"
  machine_type = "n1-standard-1"
  zone         = "us-central1-a"
}
```

Providers: Explicit vs Implicit Configuration

- Implicit: No configuration required for default provider
- Explicit: Custom provider configuration using aliases
- By default, if you define a provider block without an alias, it applies to all resources from that provider.
- But, you can use aliases to explicitly define multiple provider instances.

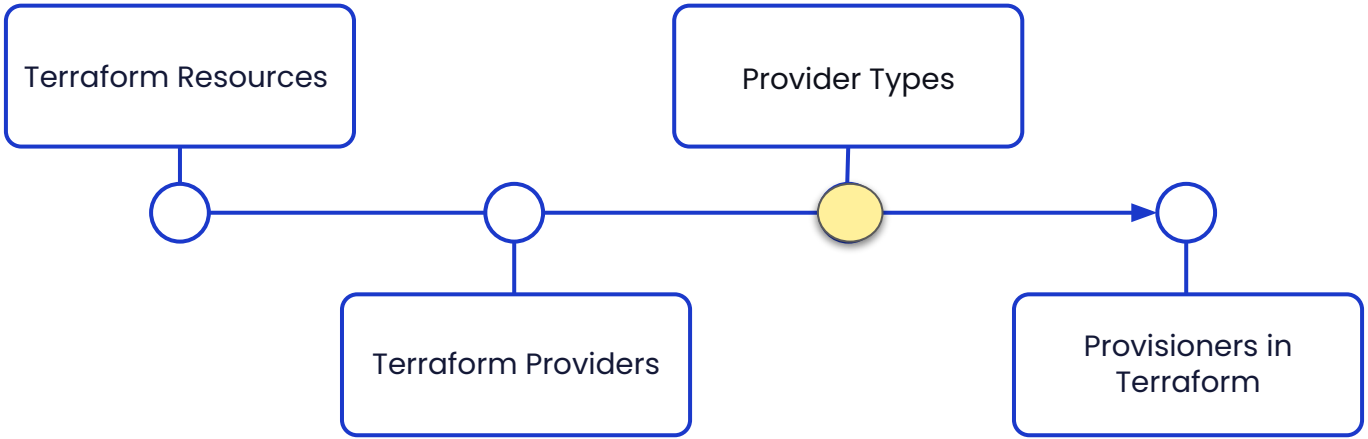
```
provider "aws" {
  region = "us-west-2"
}

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

resource "aws_instance"
"west_instance" {
  provider = aws
  ami           = "ami-123456"
  instance_type = "t2.micro"
}

resource "aws_instance"
"east_instance" {
  provider = aws.east
  ami           = "ami-123456"
  instance_type = "t2.micro"
}
```

Managing Providers with Terraform Registry

- Providers are sourced from the Terraform Registry
- Version control for providers
- Community and official providers available
- To manage provider versions, Terraform allows you to set version constraints in the provider block:

```
provider "aws" {
  version = "~> 3.0"
  region  = "us-west-2"
}
```

Terraform


Terraform Resources
Provider Types
Terraform Providers
Provisioners in Terraform

AWS Provider: Detailed Example

- Configure AWS provider
- Create an EC2 instance
- Add an S3 bucket with versioning

```
provider "aws" {
  region = "us-west-2"
}

resource "aws_instance" "example"
{
  ami           = "ami-123456"
  instance_type = "t2.micro"
}

resource "aws_s3_bucket"
"example" {
  bucket = "my-versioned-bucket"
  versioning {
    enabled = true
  }
}
```

Azure Provider: Detailed Example

- Configure Azure provider
- Create a virtual machine
- Deploy a managed disk and public IP

```
provider "azurerm" {
  features {}
}

resource "azurerm_virtual_network" "example" {
  name                = "example-vnet"
  address_space       = ["10.0.0.0/16"]
  location            = "East US"
  resource_group_name = "example-resources"
}

resource "azurerm_public_ip" "example" {
  name                = "example-pip"
  location            = "East US"
  resource_group_name = "example-resources"
  allocation_method   = "Dynamic"
}
```

GCP Provider: Detailed Example

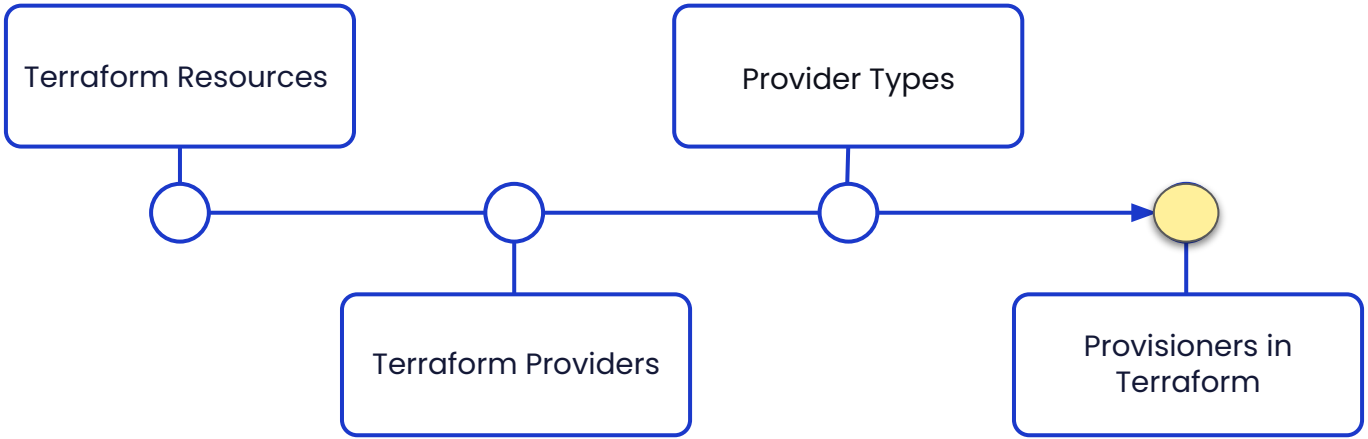
- Configure GCP provider
- Create a Compute Engine instance
- Set up a Cloud Storage bucket

```
provider "google" {
  project = "my-project"
  region  = "us-central1"
}

resource "google_compute_instance"
"example" {
  name         = "terraform-instance"
  machine_type = "n1-standard-1"
  zone         = "us-central1-a"

  boot_disk {
    initialize_params {
      image = "debian-cloud/debian-9"
    }
  }

  network_interface {
    network = "default"
    access_config {}
  }
}
```

Custom Providers in Terraform

- Use third-party or community providers
- Creating custom providers for internal use
- Examples: GitHub, Datadog, Kubernetes

Terraform


Terraform Resources
Provider Types
Terraform Providers
Provisioners in Terraform

What Are Provisioners in Terraform?

- Execute scripts or commands on resources
- Used for bootstrapping, configuration, or clean-up
- Provisioners run during create, update, or destroy
- Example: local-exec, remote-exec

Provisioners in Resources

- Execute scripts on resources during creation or destruction
- Use cases: configuring servers, running commands
- Example: SSH provisioner on EC2 instance

```
resource "aws_instance" "example" {
  ami           = "ami-123456"
  instance_type = "t2.micro"

  provisioner "remote-exec" {
    inline = [
      "sudo apt-get update",
      "sudo apt-get install -y nginx"
    ]

    connection {
      type     = "ssh"
      user     = "ubuntu"
      private_key = file("~/.ssh/my-key")
      host     = self.public_ip
    }
  }
}
```

Best Practices for Resources and Providers

- Use version pinning for stability
- Modularize infrastructure for reusability
- Store sensitive information securely

```
resource "aws_instance" "example" {
  ami           = "ami-123456"
  instance_type = "t2.micro"

  provisioner "remote-exec" {
    inline = [
      "sudo apt-get update",
      "sudo apt-get install -y nginx"
    ]

    connection {
      type     = "ssh"
      user     = "ubuntu"
      private_key = file("~/.ssh/my-key")
      host     = self.public_ip
    }
  }
}
```

Summary

1. **Terraform Resources**: Resources are the fundamental building blocks in Terraform, representing infrastructure components like servers, databases, and storage. Each resource belongs to a specific provider and is declared using a resource block.

2. **Terraform Providers**: Providers are plugins that allow Terraform to interact with external APIs such as AWS, Azure, and GCP. They define the resources Terraform can manage, and multiple providers can be used in a single configuration.

3. **Resource Configuration**: Resource blocks contain both required and optional arguments, allowing for customization. Terraform supports lifecycle management, providing control over actions like creation, update, and deletion of resources.

4. **Best Practices**: Ensure stability by pinning provider versions, modularizing infrastructure for reusability, and securely storing sensitive information like API keys using environment variables or secret management systems.

Digital Career Institute

Next up

Terraform | Week 1

DAY 3:
Terraform State Management

LEARN FOR **A NEW LIFE.**

DCI