TERRAFORM - INFRASTRUCTURE AS CODE

THE PROBLEM: MULTI-CLOUD CHALLENGE

SCENARIO: FLIPKART EXAMPLE

Company: Flipkart

Situation: Flipkart as a company can create their compute resources anywhere.

What are compute resources? Compute resources are CPU, RAM, storage, etc.

Deployment options:

- AWS (Amazon Web Services)
- Azure (Microsoft Azure)
- GCP (Google Cloud Platform)
- On-premises (their own physical servers)

COMPANY NEEDS

Number of applications: 300 applications

Requirement: To deploy these 300 apps, they need servers.

Where can they deploy:

- AWS
- Azure
- GCP
- Physical servers (on-premises)

INITIAL DECISION: AWS

As a DevOps engineer, you decide: Let's deploy all on AWS

This decision went well initially: You started creating:

- EC2 instances

- S3 buckets
- RDS databases
- VPCs
- Security groups
- And other AWS resources

AUTOMATION ON AWS

Problem: Creating resources manually through AWS console is time-consuming.

Solution options:

Option 1: AWS CLI Write scripts using AWS command-line interface

Option 2: AWS CloudFormation Template (CFT) Use AWS CloudFormation to automate infrastructure

Your decision: You used CFT and automated infrastructure on AWS.

CLOUDFORMATION SUCCESS

Example scenario:

Developer request: "I want 10 EC2 instances on AWS"

Your response: "Using CFT, I wrote a script. I will execute the script and 10 instances will be generated in no time."

What you automated:

- Created scripts for EC2 instances
- Created scripts for S3 buckets
- Created scripts for RDS databases
- All AWS resources automated

Everything is working perfectly on AWS

THE BIG PROBLEM: VENDOR LOCK-IN

Company announces: Due to some reasons, we need to shift to another cloud platform.

Reasons could be:

- Cost optimization
- Better services on other platform
- Compliance requirements
- Business decisions
- Vendor relationship changes

Your situation now: All these scripts will be of no use.

Why? We cannot use CFT (CloudFormation Templates) on other cloud platforms because CFT belongs to AWS cloud provider only.

THE CHALLENGE

If you want to move to:

Option 1: Azure

- Cannot use AWS CFT
- Need to learn Azure Resource Manager (ARM)
- Rewrite all scripts in ARM templates

Option 2: GCP

- Cannot use AWS CFT or Azure ARM
- Need to learn GCP Deployment Manager
- Rewrite all scripts again

Option 3: On-premises (OpenStack)

- Cannot use CFT, ARM, or GCP tools
- Need to learn Heat templates (OpenStack)
- Rewrite everything again

Problem: Migrating scripts from one cloud platform to another and to Heat templates (in case of on-premises) is very hectic.

HYBRID CLOUD REALITY

What is hybrid cloud? Company hosts part of infrastructure on AWS and part on Azure (or multiple cloud platforms).

Example:

- Storage: Using AWS S3
- Project management and build services: Using Azure DevOps
- Databases: Using GCP Cloud SQL
- On-premises: Legacy applications

Your challenge as DevOps engineer: You need to learn:

- AWS CFT for AWS automation
- Azure ARM for Azure automation
- GCP Deployment Manager for GCP automation
- Heat templates for on-premises
- Different syntax for each
- Different concepts for each

This is impractical and very difficult!

THE SOLUTION: TERRAFORM

To avoid all these problems of learning too many things, we use Terraform.

What is Terraform? Terraform is developed by HashiCorp.

HashiCorp's promise: "DevOps engineers, don't learn 100 tools. Just learn one tool called Terraform."

HOW TERRAFORM SOLVES THE PROBLEM

Terraform's capability: If you are a DevOps engineer, you can create Terraform scripts and Terraform will take care of automating the resources on any cloud platform.

Process:

Step 1: Tell Terraform that your provider is AWS

Step 2: Write scripts related to AWS resources

Step 3: If you want to switch from AWS to Azure:

- Make very minimum changes
- Change provider from AWS to Azure
- Modify a few resource-specific details
- Everything else remains similar

Step 4: Just looking at Terraform documentation, you can write automation for any infrastructure.

THE CONCEPT: INFRASTRUCTURE AS CODE (IaC)

What is IaC? Infrastructure as Code is writing infrastructure configuration in code files.

Tools for IaC:

- AWS CloudFormation (AWS only)
- Azure Resource Manager (Azure only)
- GCP Deployment Manager (GCP only)
- Terraform (Multi-cloud)

Problem with cloud-specific tools: Each cloud has different IaC tool. You need to learn multiple tools.

Terraform advantage: One tool for all clouds.

THE ADVANCED CONCEPT: API AS CODE

What is API as Code? Terraform works on the concept called API as Code.

This is a concept using which we can automate any provider:

- AWS
- Azure
- GCP
- On-premises cloud services

How? Using their APIs.

UNDERSTANDING APIs

What is API? API = Application Programming Interface

Definition: API is a way to programmatically talk to any application.

EXAMPLE: GOOGLE

Manual way (UI):

1. You want to talk to Google
2. Open your laptop
3. Open a browser
4. Type www.google.com
5. Press Enter
6. You send a request to Google
7. You get a page where you can search information

This is a UI (User Interface) approach

Programmatic way (API):

Scenario: You want to get some information from Google in a programmatic way.

Problem with UI approach:

- Not suitable for automation
- Cannot run scripts
- Manual steps required

Solution with API: Instead of logging into laptop, opening browser, and searching:

1. Write a script
2. Script talks to Google API
3. API executes action
4. Sends result back

No manual intervention needed

HOW APIs WORK

Developers' solution: To perform this kind of automation, developers have come up with a concept called API.

Using API: You can programmatically talk to any application.

GITHUB API EXAMPLE

GitHub approach:

Manual way:

1. Open browser
2. Go to github.com
3. Login with username and password
4. Navigate to repository
5. View files

API way: GitHub will expose their API.

GitHub says: "Instead of manually logging into GitHub and authenticating with me, you can:

1. Open terminal
2. Use curl request or HTTP GET request
3. Talk to my API
4. Get what information you want
5. I will send the response"

Example API call:

```
curl -H "Authorization: token YOUR_TOKEN" https://api.github.com/repos/owner/repo
```

Response: JSON data with repository information

TWO WAYS TO INTERACT WITH APPLICATIONS

1. Manually using User Interface (UI):

- Login through browser
- Click buttons
- Visual interface

- Good for humans
- Not good for automation

2. Programmatically using API:

- Write scripts
- Make API calls
- Get JSON/XML responses
- Good for automation
- Efficient

The programmatic interface here is API.

HOW TERRAFORM USES APIs

Terraform uses the same concept of APIs.

TRADITIONAL APPROACH (Without Terraform):

Scenario: Create EC2 instance on AWS

Steps:

1. Learn AWS API documentation
2. Understand API endpoints
3. Write code to make API call
4. Handle authentication
5. Parse JSON response
6. Handle errors

Example API call to create EC2:

python

```
import boto3


ec2 = boto3.client('ec2')
response = ec2.run_instances(
  ImageId='ami-12345678',
  InstanceType='t2.micro',
  MinCount=1,
  MaxCount=1,
  KeyName='mykey',
  SecurityGroupIds=['sg-12345678'],
  SubnetId='subnet-12345678'
)
```

Problems:

- Complex code
- Need to know AWS SDK
- Different code for each cloud
- Hard to maintain

TERRAFORM APPROACH:

What Terraform does: Instead of developers talking directly to API, Terraform actively looks at these APIs and Terraform has written their own modules.

TERRAFORM WORKFLOW

Step 1: You write Terraform file

Example: Create EC2 instance

Instead of you directly making API call with AWS, Terraform says: "There is a module called EC2. Using this EC2 module, you can write a bunch of lines just like you write in English."

Terraform file (main.tf):

hcl

```
resource "aws_instance" "my_server" {
  ami         = "ami-12345678"
  instance_type = "t2.micro"
  key_name      = "mykey"

  tags = {
    Name = "MyServer"
  }
}
```

Much simpler than API code!

Step 2: You submit Terraform file

Command:
```
terraform apply
```

Once user submits this request in Terraform file, Terraform will receive this request.

Step 3: Terraform processes your input

What happens in the backend:

1. Terraform takes your input
2. Terraform reads provider configuration (AWS)

3. Terraform converts your input into API call

4. Terraform makes API call to AWS

5. AWS processes request

6. AWS creates EC2 instance

7. AWS sends response back to Terraform

8. Terraform sends response back to you

VISUAL FLOW

` ` `

User (DevOps Engineer)

↓

Writes Terraform script (main.tf)

↓

Executes: terraform apply

↓

Terraform reads script

↓

Terraform converts to API call

↓

Terraform → AWS API (create EC2 instance)

↓

AWS creates instance

↓

AWS → Terraform (success response)

↓

Terraform → User (instance created)

```

---

**KEY POINT**

**As a user:**

You are NOT directly talking to APIs of AWS or Azure.

**You are:**

Writing Terraform scripts.

**Terraform is:**

Converting your requests into API calls.

**This is the concept of Terraform.**

**This is how Terraform utilizes the concept called API as Code.**

---

**INFRASTRUCTURE AS CODE (IaC) vs API AS CODE**

**Infrastructure as Code (IaC):**

A concept using which you can automate your infrastructure.

**Examples:**

- AWS CloudFormation
```

- Azure Resource Manager

- GCP Deployment Manager

Limitation:

Each tool works only with one cloud provider.

API as Code:

Terraform works on the same concept (IaC) with advanced capability called API as Code.

Terraform's advantage:

Just learn one language called Terraform, and Terraform will look after everything.

TERRAFORM BENEFITS

Benefit 1: Single language

Learn Terraform syntax once, use everywhere

Benefit 2: Multi-cloud support

Same tool for AWS, Azure, GCP, on-premises

Benefit 3: Provider abstraction

Terraform talks to provider APIs, you write simple code

Benefit 4: Minimal code changes

Switching cloud providers requires minimal changes

Benefit 5: Open source

Free to use, large community

Benefit 6: State management

Terraform tracks what was created

Benefit 7: Plan before apply

See what will change before making changes

TERRAFORM ARCHITECTURE

```
Terraform Core
   ↓
Reads Terraform files (.tf)
   ↓
Understands provider (AWS/Azure/GCP)
   ↓
Converts to provider API calls
   ↓
Provider APIs
    ├── AWS API
    ├── Azure API
    ├── GCP API
```

```
  └── Other APIs
  ↓
Cloud providers create resources
  ↓
Response back to Terraform
  ↓
Terraform updates state
  ↓
Shows results to user
```

EXAMPLE: SWITCHING CLOUD PROVIDERS

AWS Terraform file:

provider.tf:

hcl

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

main.tf:

hcl

```
resource "aws_instance" "web_server" {
  ami           = "ami-12345678"
  instance_type = "t2.micro"

  tags = {
    Name = "WebServer"
  }
}
```

Switch to Azure with minimal changes:

provider.tf:

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

main.tf:

```hcl
resource "azurerm_virtual_machine" "web_server" {
  name            = "WebServer"
  location         = "East US"
  vm_size          = "Standard_B1s"

  # Other Azure-specific configurations
}
```

Changes needed:

- Provider block (aws → azurerm)
- Resource type (aws_instance → azurerm_virtual_machine)
- Resource-specific parameters

Core concept remains same:

- resource keyword
- resource name
- configuration block
- Same Terraform commands

TERRAFORM WORKFLOW

Step 1: Write Terraform files
```
provider.tf - Define cloud provider

main.tf - Define resources

variables.tf - Define variables

outputs.tf - Define outputs
```

Step 2: Initialize Terraform
```
terraform init
```

Downloads provider plugins

Step 3: Plan changes
```
terraform plan
```

Shows what will be created/changed/destroyed

Step 4: Apply changes
```
```

```
terraform apply
```

Creates actual resources

Step 5: Verify

Check cloud console, resources created

Step 6: Destroy (when needed)

```
terraform destroy
```

Removes all created resources

DETAILED EXAMPLE: AWS vs AZURE

Task: Create virtual machine

AWS version:

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

resource "aws_instance" "my_vm" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
  key_name      = "mykey"

  tags = {
    Name        = "MyVM"
    Environment = "Dev"
  }
}

resource "aws_s3_bucket" "my_bucket" {
  bucket = "my-unique-bucket-name"

  tags = {
    Name = "MyBucket"
  }
}
```

Run:
```
terraform init
terraform plan
terraform apply
```
Result: EC2 instance and S3 bucket created on AWS

Azure version (switching providers):

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

resource "azurerm_resource_group" "my_rg" {
  name     = "myResourceGroup"
  location = "East US"
}

resource "azurerm_virtual_machine" "my_vm" {
  name                = "MyVM"
  location            = azurerm_resource_group.my_rg.location
  resource_group_name = azurerm_resource_group.my_rg.name
  vm_size             = "Standard_B1s"

  # Additional configuration...

  tags = {
    Name        = "MyVM"
    Environment = "Dev"
  }
}

resource "azurerm_storage_account" "my_storage" {
  name                     = "myuniquestorage"
  resource_group_name      = azurerm_resource_group.my_rg.name
  location                 = azurerm_resource_group.my_rg.location
  account_tier             = "Standard"
  account_replication_type = "LRS"
}
```

Run:

```
```

terraform init

terraform plan

terraform apply

Result: Virtual machine and storage account created on Azure

WHAT CHANGED:

Changed:

- Provider name (aws → azurerm)
- Resource types (aws_instance → azurerm_virtual_machine)
- Some parameter names

Remained same:

- Terraform commands (init, plan, apply)
- Overall syntax structure
- Workflow
- Concepts

You made very minimum changes and things are taken care of.

HYBRID CLOUD EXAMPLE

Scenario: Company uses both AWS and Azure

Single Terraform project:

provider.tf:

hcl

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


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

main.tf:

```hcl
# AWS resources
resource "aws_s3_bucket" "storage" {
  bucket = "my-aws-bucket"
}

resource "aws_instance" "web" {
  ami           = "ami-12345678"
  instance_type = "t2.micro"
}

# Azure resources
resource "azurerm_resource_group" "rg" {
  name     = "myResourceGroup"
  location = "East US"
}

resource "azurerm_storage_account" "storage" {
  name                = "myazurestorage"
  resource_group_name = azurerm_resource_group.rg.name
  location            = azurerm_resource_group.rg.location
}
```

Result: One Terraform project manages both AWS and Azure resources!