Terraform Workshop

How to create your infrastructure as code

In today's workshop...

What is Terraform?

Is it really that good?

Quick Example

How to create an S3 Bucket

10 min

45min

Basic Knowledge

Resources, Data Sources, Providers, Variables and more

Exercises

How to create your immutable infrastructure

Who created it?

A company called Hashicorp.

https://www.terraform.io

Written in Golang to deliver Infrastructure as Code.

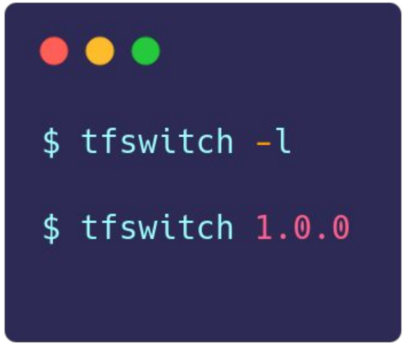
We can plan our changes and store them in a Terraform State File.

How to install Terraform

I recommend using a tool called TfSwitch.

https://tfswitch.warrensbox.com/

It’s a Golang binary that downloads a specific Terraform version, since Hashicorp is always updating it.



```
$ tfswitch -l

$ tfswitch 1.0.0
```

How Terraform keeps everything?

State File

Terraform must store state about your managed infrastructure and configuration. This state is used by Terraform to map real world resources to your configuration, keep track of metadata, and to improve performance for large infrastructures.

This state is stored by default in a local file named "terraform.tfstate", but it can also be stored remotely, which works better in a team environment.

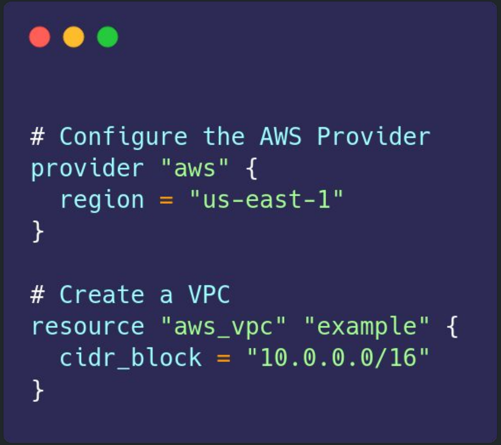
https://www.terraform.io/docs/language/state/index.html

Terraform Concepts

Providers

Terraform Providers are plugins(official or not) that interact with cloud providers or other APIs.

Each provider sets a series of resources and data sources for us to use.



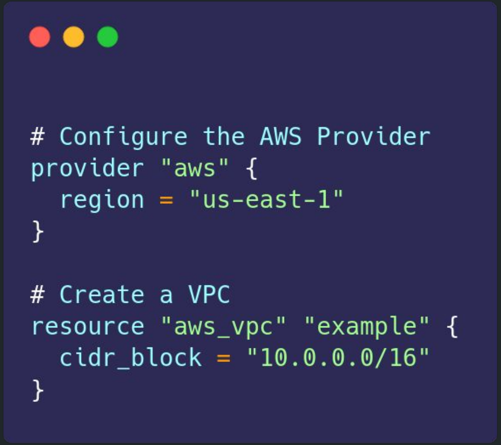
```
# Configure the AWS Provider
provider "aws" {
  region = "us-east-1"
}

# Create a VPC
resource "aws_vpc" "example" {
  cidr_block = "10.0.0.0/16"
}
```

https://registry.terraform.io/browse/providers

Resources

Terraform Resources describe one or more infrastructure objects.

This block of code will create anything in the provider.

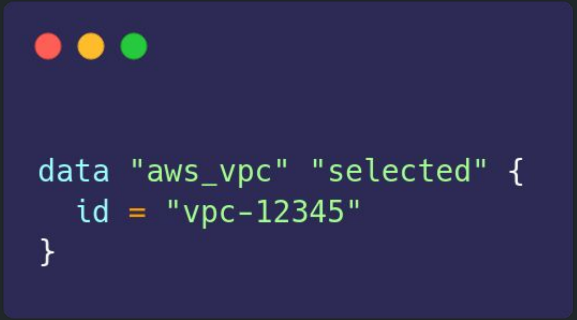


```
# Configure the AWS Provider
provider "aws" {
  region = "us-east-1"
}

# Create a VPC
resource "aws_vpc" "example" {
  cidr_block = "10.0.0.0/16"
}
```

https://registry.terraform.io/providers/hashicorp/aws/latest/docs

Data Sources

Data sources allow Terraform use information defined outside of Terraform, defined by another separate Terraform configuration, or modified by functions.

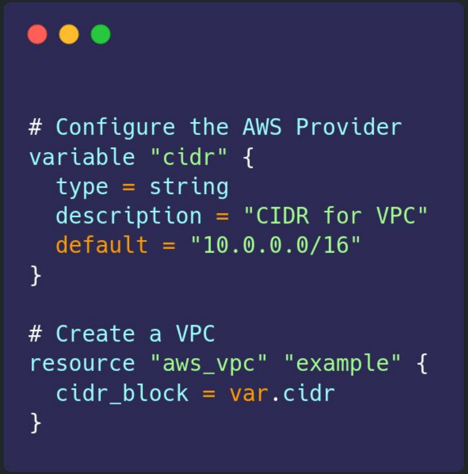


```
data "aws_vpc" "selected" {
  id = "vpc-12345"
}
```

https://www.terraform.io/docs/language/data-sources/index.html

Variables

Serve as parameters for Terraform, so you can customize its behavior in different scenarios.

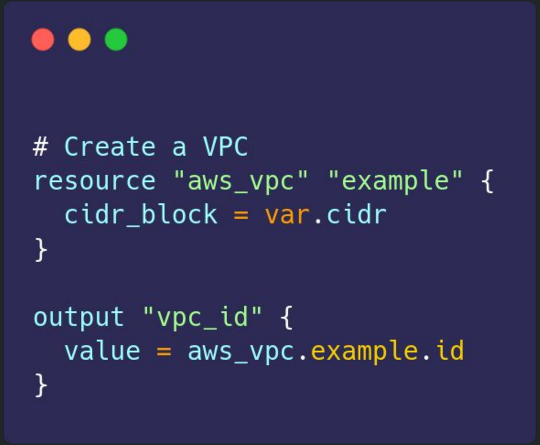


```
# Configure the AWS Provider
variable "cidr" {
  type = string
  description = "CIDR for VPC"
  default = "10.0.0.0/16"
}

# Create a VPC
resource "aws_vpc" "example" {
  cidr_block = var.cidr
}
```

https://www.terraform.io/docs/language/values/index.html

Outputs

Outputs are like return values for a Terraform module.



```
# Create a VPC
resource "aws_vpc" "example" {
  cidr_block = var.cidr
}

output "vpc_id" {
  value = aws_vpc.example.id
}
```

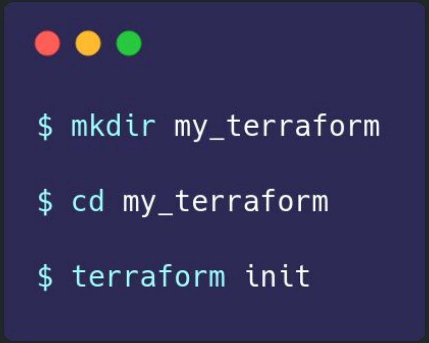
https://www.terraform.io/docs/language/values/index.html

Let’s create something...

- Using the AWS Provider
- AWS SQS Queue
- Tags: Name and Environment
- Using Variables
- Output Queue URL
- Destroy everything

Init

This command is used to initialize a working directory containing Terraform configuration files.



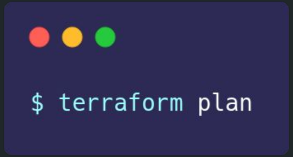
```
$ mkdir my_terraform

$ cd my_terraform

$ terraform init
```

https://www.terraform.io/docs/cli/commands/init.html

Plan

This command creates an execution plan. By default, creating a plan consists of:

- Reading the current state of any already-existing remote objects to make sure that the Terraform state is up-to-date.
- Comparing the current configuration to the prior state and noting any differences.
- Proposing a set of change actions that should, if applied, make the remote objects match the configuration.



```
$ terraform plan
```

https://www.terraform.io/docs/cli/commands/plan.html

Apply

The terraform apply command executes the actions proposed in a Terraform plan.

If the flag *auto-approve* has **not** been used, you'll need to manually approve all the changes Terraform will do.

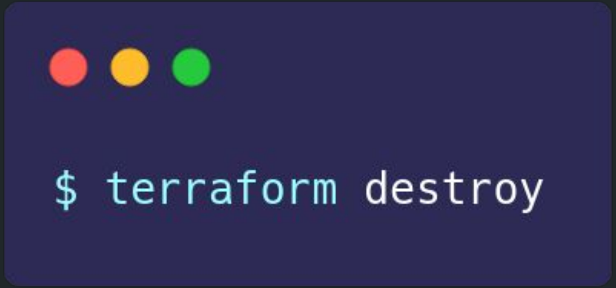
```
$ terraform apply
```

https://www.terraform.io/docs/cli/commands/apply.html

Destroy

The terraform destroy command is a convenient way to destroy all remote objects managed by a particular Terraform configuration.

If the flag auto-approve has not been used, you'll need to manually approve all the changes Terraform will do.



```
$ terraform destroy
```

https://www.terraform.io/docs/cli/commands/destroy.html

Exercices Preparation

Make sure you did this before you start your Terraform tasks

- ➔ **AWS Dev Account Access**
  Make sure you have the right access to it.
- ➔ **Terraform**
  Install TfSwitch and use version 1.0.0.
- ➔ **Open the Docs**
  Open Terraform official documentation to help you during the tasks.

General Rules

- ➔ You cannot use Terraform Modules.
- ➔ Destroy everything in the end of each exercise.
- ➔ Share what you have learned today.

First Exercise

1. Create a S3 Bucket
2. Use Tags -> Name, Owner and Environment
3. Enable Versioning
4. Output the Bucket ID and all of its tags
5. Describe the aws_s3_bucket resource using Terraform.
6. Destroy everything

Second Exercise

1. Create an EC2 Instance on a Public Subnet(don't create any network resources, you need to use what has already been created)
2. Use Tags -> Name, Owner and Environment
3. Install Nginx on boot time
4. Create a custom index.html with the message "Hello. This is my random number <n>" and replace the default index. Also on boot time.
5. Make sure you can access your instance only on port 80. You cannot SSH into your Instance.
6. Output the Instance Public IP.
7. Destroy everything

Third Exercise

1. Using the last exercice as a starting point
2. Create an AutoScaling Group for that Instance
3. Use an Application Load Balancer on port 80
4. Spawn 3 instances at minimum
5. Use a *.tfvars* file for your variables
6. Output the ALB DNS
7. Destroy everything