Introduction

Infrastructure as a Code:

Infrastructure as Code (IaC) uses a high-level descriptive coding language to automate the provisioning of IT infrastructure. This automation eliminates the need for developers to manually provision and manage servers, operating systems, database connections, storage, and other infrastructure elements every time they want to develop, test, or deploy a software application.

Infrastructure as Code benefits

Provisioning traditional IT is a time-consuming and costly process, requiring the physical setup of the hardware, installation and configuration of operating system software, and connection to middleware, networks, storage, etc. by expert personnel.

Virtualization and cloud native development eliminate the problem of physical hardware management, enabling developers to provision their own virtual servers or containers on demand. But, provisioning virtualized infrastructure still distracts developers' focus from coding, still requires them to repeat provisioning work for every new deployment, and doesn't provide an easy way to track environment changes and prevent inconsistencies that impact deployments.

Infrastructure as Code (IaC) goes the final step of enabling developers to effectively 'order up' fully documented, versioned infrastructure by executing a script. The benefits are exactly what you might imagine:

- **Faster time to production/market:** IaC automation dramatically speeds the process of provisioning infrastructure for development, testing, and production (and for scaling or taking down production infrastructure as needed). Because it codifies and documents everything, IaC can even automate provisioning of legacy infrastructure, which might otherwise be governed by time-consuming processes (like pulling a ticket).
- **Improved consistency**—less 'configuration drift': Configuration drift occurs when ad-hoc configuration changes and updates result in a mismatched development, test, and deployment environments. This can result in issues at deployment, security vulnerabilities, and risks when developing applications and services that need to meet strict regulatory compliance standards. IaC prevents drift by provisioning the same environment every time.
- **Faster, more efficient development**: By simplifying provisioning and ensuring infrastructure consistency, IaC can confidently accelerate every phase of the software delivery lifecycle. Developers can quickly provision sandboxes and continuous integration/continuous deployment (CI/CD) environments. QA can quickly provision full-fidelity test environments. Operations can quickly provision infrastructure for security and user-acceptance testing. And when the code passes testing, the application and the production infrastructure it runs on can be deployed in one step.
- **Protection against churn**: To maximize efficiency in organizations without IaC, provisioning is typically delegated a few skilled engineers or IT staffers. If one of these specialists leaves the organization, others are sometimes left to reconstruct the process. IaC ensures that provisioning intelligence always remains with the organization.
- **Lower costs and improved ROI:** In addition to dramatically reducing the time, effort, and specialized skill required to provision and scale infrastructure, IaC lets

organizations take maximum advantage of cloud computing's consumption-based cost structure. It also enables developers to spend less time on plumbing and more time developing innovative, mission-critical software solutions.

We have two ways of infrastructure automation

1. Imperative
2. Declarative

Declarative vs. imperative approach

When choosing an IaC solution, it's also important to understand the difference between a declarative or an imperative approach to infrastructure automation.

In most organizations, the declarative approach—also known as the functional approach—is the best fit. In the declarative approach, you specify the desired final state of the infrastructure you want to provision and the IaC software handles the rest—spinning up the virtual machine (VM) or container, installing and configuring the necessary software, resolving system and software interdependencies, and managing versioning. The chief downside of the declarative approach is that it typically requires a skilled administrator to set up and manage, and these administrators often specialize in their preferred solution. Eg: Terraform.

In the imperative approach—also known as the procedural approach—the solution helps you prepare automation scripts that provision your infrastructure one specific step at a time. While this can be more work to manage as you scale, it can be easier for existing administrative staff to understand and can leverage configuration scripts you already have in place.

Simple analogy:

When we wanted to drive a car from point A to point B we'll follow a set of instruction(Go left, Take right...) this is kind of an imperative approach to automation. What terraform does is imagine like you took an uber to go from point A to point B. We don't have to worry about the Instructions anymore we just need to say to driver that we need to go to point B.

Terraform

Terraform is an infrastructure provisioning tool created by Hashicorp. It allows you to describe your infrastructure as code, creates "execution plans" that outline exactly what will happen when you run your code, builds a graph of your resources, and automates changes with minimal human interaction.

Terraform uses its own domain-specific language (DSL) called Hashicorp Configuration Language (HCL). HCL is JSON-compatible and is used to create these configuration files that describe the infrastructure resources to be deployed.

Terraform is cloud-agnostic and allows you to automate infrastructure stacks from multiple cloud service providers simultaneously and integrate other third-party services.

Simple NGINX server

Process

For a terraform to deploy any infrastructure it has three stages.

1. **CODE** : In this we'll actually write the code up that terraform file. As we have three resources we'll include all the three resources in the terraform file along with the arguments like name, networking and data center.
2. **PLAN**: This is actually a terraform command in the TF CLI we run this command. What's it going to do id it's going to compare the desired state to what actually exists and it says what need to be created for desired state.
3. **APPLY** : Terraform is going to work against the cloud providers using real API's and our API Token to spin up these infrastructure resources and its gonna output some autogenerated variables along the way for may be kubernettes dashboard URL or URL to access our application.

If you have Terraform and Docker ins
Lets try to build an Nginx webserver and view the default Ngnix page

CODE:

we need to configure the required resources in the main.tf file.

Create a file named **Main.tf** and write down the required infrastructure in this case we need a docker image and a docker container.

```
resource "docker_image" "nginx" {
  name = "nginx:latest"
}

resource "docker_container" "nginx" {
  image = docker_image.nginx.latest
  name  = "naveen"
  ports {
    internal = 80
    external = 80
  }
}
```

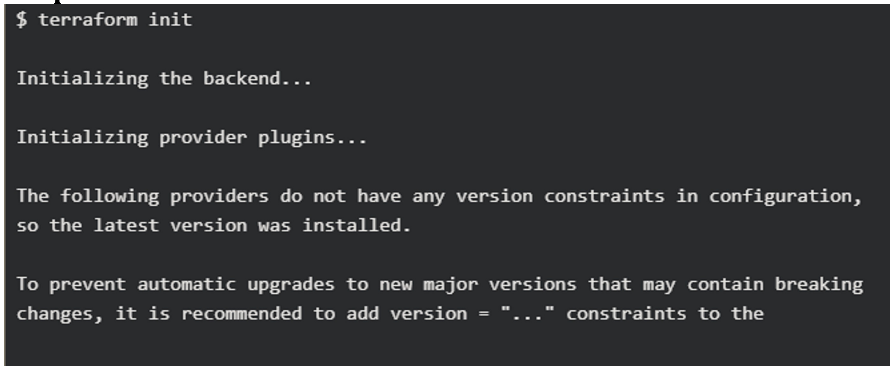
It defines two resources: a Docker disk image that packages the Nginx webserver, and a Docker container that gives it a name and runs it on port 80.

Init

All Terraform workflows start with the init command. Terraform searches the configuration for both direct and indirect references to providers (such as Docker). Terraform then attempts to load the required plugins.

terraform init

Output:



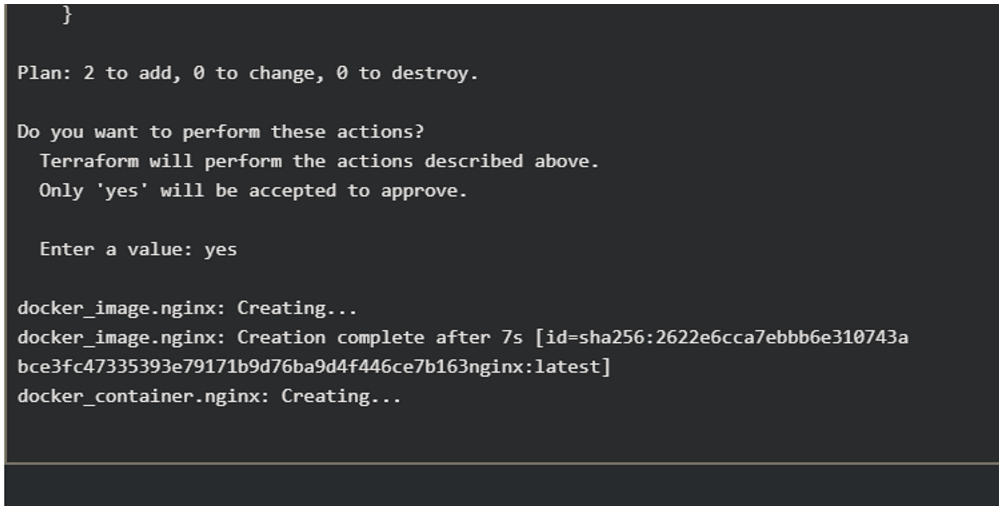
```
$ terraform init

Initializing the backend...

Initializing provider plugins...

The following providers do not have any version constraints in configuration,
so the latest version was installed.

To prevent automatic upgrades to new major versions that may contain breaking
changes, it is recommended to add version = "..." constraints to the
```

A**pply**

Now provision the webserver by running apply.

terraform apply

You will be asked to confirm. Type yes and press ENTER. A message will display confirmation that it succeeded.



```
    }

Plan: 2 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes

docker_image.nginx: Creating...
docker_image.nginx: Creation complete after 7s [id=sha256:2622e6cca7ebbb6e310743a
bce3fc47335393e79171b9d76ba9d4f446ce7b163nginx:latest]
docker_container.nginx: Creating...
```

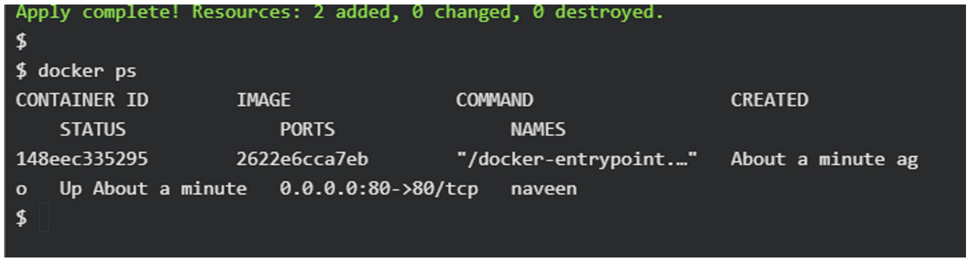
Verify

Visit this URL to view the default Nginx web page which is now live:

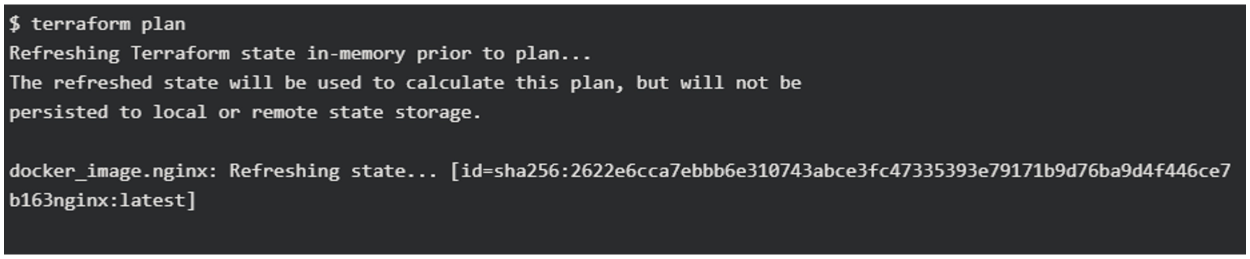
Nginx index page

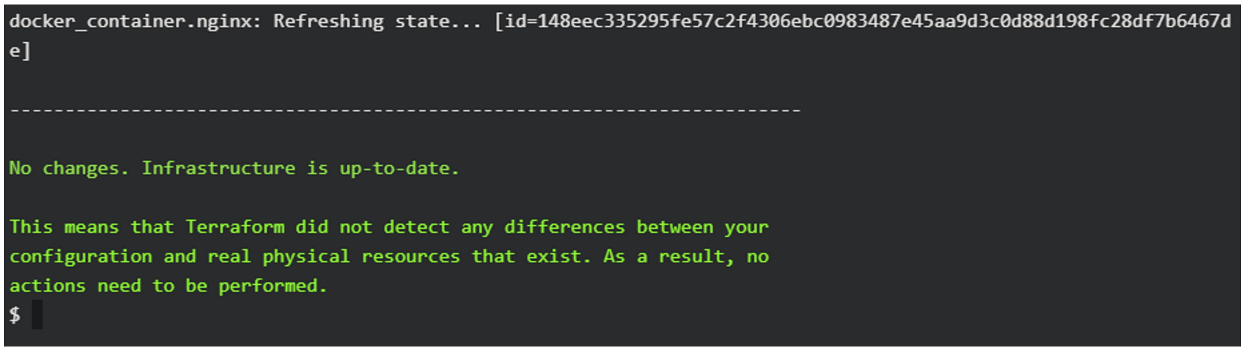
Alternatively, you can examine Docker's process list. You will see the tutorial container which is running Nginx.

docker ps


```
Apply complete! Resources: 2 added, 0 changed, 0 destroyed.
$
$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED
    STATUS              PORTS               NAMES
148eec335295        2622e6cca7eb        "/docker-entrypoint.…"   About a minute ag
o   Up About a minute   0.0.0.0:80->80/tcp   naveen
$
```

Without making any changes I'm trying to run terraform plan


```
$ terraform plan
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but will not be
persisted to local or remote state storage.

docker_image.nginx: Refreshing state... [id=sha256:2622e6cca7ebbb6e310743abce3fc47335393e79171b9d76ba9d4f446ce7
b163nginx:latest]
```


```
docker_container.nginx: Refreshing state... [id=148eec335295fe57c2f4306ebc0983487e45aa9d3c0d88d198fc28df7b6467d
e]

------------------------------------------------------------------------

No changes. Infrastructure is up-to-date.

This means that Terraform did not detect any differences between your
configuration and real physical resources that exist. As a result, no
actions need to be performed.
$
```

Since we haven't made any changes to the terraform file the desired state is similar to the actual state.

To stop the container, run terraform destroy.

$ terraform destroy

Inspect state
When you applied your configuration, Terraform wrote data into a file called terraform.tfstate. This file now contains the IDs and properties of the resources Terraform created so that it can manage or destroy those resources going forward.

You must save your state file securely and distribute it only to trusted team members who need to manage your infrastructure. In production, we recommend storing your state remotely. Remote stage storage enables collaboration using Terraform but is beyond the scope of this tutorial.

We can inspect the current state using **terraform show**

terraform validate

If you are copying configuration snippets or just want to make sure your configuration is syntactically valid and internally consistent, the built in **terraform validate** command will check and report errors within modules, attribute names, and value types.

Validate your configuration. If your configuration is valid, Terraform will return a success message. installed on your local machine, start Docker Desktop.

$ open -a Docker

Build AWS instance using Terraform

We'll be trying to create an AWS instance from terraform.

Pre-requisites:

1. Setting up AWS account -> aws Signup
2. AWS CLI installed and the credentials configured locally.

Code:

The set of files used to describe infrastructure in Terraform is known as a Terraform configuration. You'll write your first configuration now to launch a single AWS EC2 instance.

Each configuration should be in its own directory. Create a directory for the new configuration and create a file as **configure.ts**

```
provider "aws" {
  profile = "default"
  region  = var.region
}
resource "aws_instance" "nemnous" {
  ami           = "ami-0a0ddd875a1ea2c7f"
  instance_type = "t2.micro"
  provisioner "local-exec" {
    command = "echo ${aws_instance.nemnous.public_ip} > ip_address.txt"
  }
}
resource "aws_eip" "ip" {
  vpc      = true
  instance = aws_instance.nemnous.id
}
output "ip" {
  value = aws_eip.ip.public_ip
}
```

Provider:

The provider block configures the named provider, in our case aws, which is responsible for creating and managing resources. A provider is a plugin that Terraform uses to translate the API interactions with the service. A provider is responsible for understanding API interactions and exposing resources. Because Terraform can interact with any API, you can represent almost any infrastructure type as a resource in Terraform.

The profile attribute in your provider block refers Terraform to the AWS credentials stored in your AWS Config File, which you created when you configured the AWS CLI. HashiCorp recommends that you never hard-code credentials into *.tf configuration files. We are explicitly defining the default AWS config profile here to illustrate how Terraform should access sensitive credentials.

Note: If you leave out your AWS credentials, Terraform will automatically search for saved API credentials (for example, in ~/.aws/credentials) or IAM instance profile credentials. This is cleaner when .tf files are checked into source control or if there is more than one admin user.

Multiple provider blocks can exist if a Terraform configuration manages resources from different providers. You can even use multiple providers together. For example you could pass the ID of an AWS instance to a monitoring resource from DataDog.

Resource

The resource block defines a piece of infrastructure. A resource might be a physical component such as an EC2 instance, or it can be a logical resource such as a Heroku application.

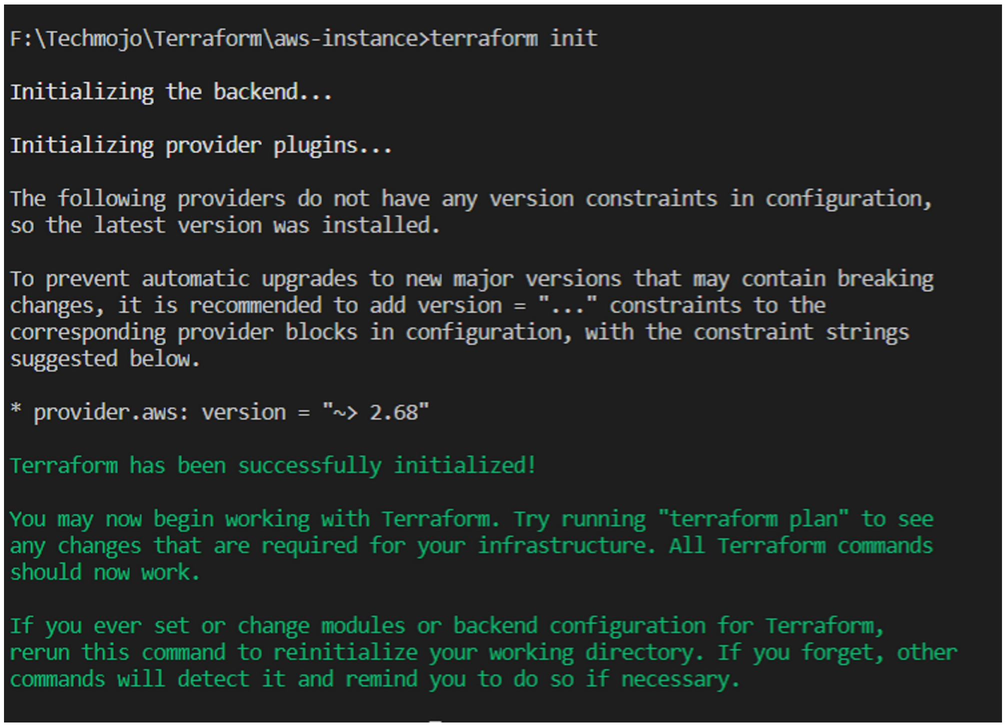
The resource block has two strings before the block: the resource type and the resource name. In the example, the resource type is aws_instance and the name is example. The prefix of the type maps to the provider. In our case "aws_instance" automatically tells Terraform that it is managed by the "aws" provider.

The arguments for the resource are within the resource block. The arguments could be things like machine sizes, disk image names, or VPC IDs. Our providers reference documents the required and optional arguments for each resource provider. For your EC2 instance, you specified an AMI for Ubuntu, and requested a t2.micro instance so you qualify under the free tier.

Initialize the directory

When you create a new configuration — or check out an existing configuration from version control — you need to initialize the directory with **terraform init**.

Terraform uses a plugin-based architecture to support hundreds of infrastructure and service providers. Initializing a configuration directory downloads and installs providers used in the configuration, which in this case is the aws provider. Subsequent commands will use local settings and data during initialization.



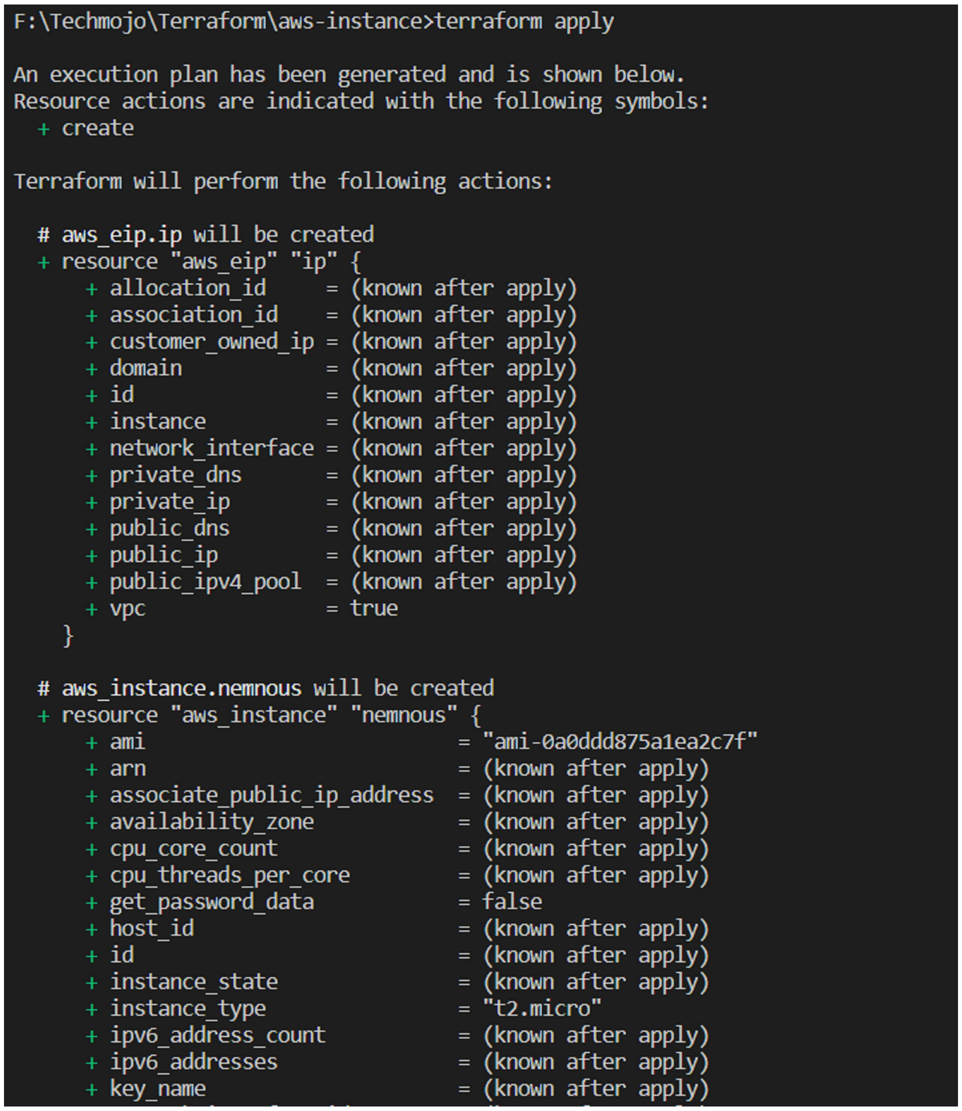
```
F:\Techmojo\Terraform\aws-instance>terraform init

Initializing the backend...

Initializing provider plugins...

The following providers do not have any version constraints in configuration,
so the latest version was installed.

To prevent automatic upgrades to new major versions that may contain breaking
changes, it is recommended to add version = "..." constraints to the
corresponding provider blocks in configuration, with the constraint strings
suggested below.

* provider.aws: version = "~> 2.68"

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
```

Terraform downloads the aws provider and installs it in a hidden subdirectory of the current working directory. The output shows which version of the plugin was installed.

Create infrastructure

In the same directory as the example.tf file you created, run terraform apply. You should see an output similar to the one shown below, though we've truncated some of the output to save space.

Note: Terraform 0.11 and earlier require running terraform plan before terraform apply. Use terraform version to confirm your running version.



```
F:\Techmojo\Terraform\aws-instance>terraform apply

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # aws_eip.ip will be created
  + resource "aws_eip" "ip" {
      + allocation_id     = (known after apply)
      + association_id    = (known after apply)
      + customer_owned_ip = (known after apply)
      + domain            = (known after apply)
      + id                = (known after apply)
      + instance          = (known after apply)
      + network_interface = (known after apply)
      + private_dns       = (known after apply)
      + private_ip        = (known after apply)
      + public_dns        = (known after apply)
      + public_ip         = (known after apply)
      + public_ipv4_pool  = (known after apply)
      + vpc               = true
    }

  # aws_instance.nemnous will be created
  + resource "aws_instance" "nemnous" {
      + ami                          = "ami-0a0ddd875a1ea2c7f"
      + arn                          = (known after apply)
      + associate_public_ip_address  = (known after apply)
      + availability_zone            = (known after apply)
      + cpu_core_count               = (known after apply)
      + cpu_threads_per_core         = (known after apply)
      + get_password_data            = false
      + host_id                      = (known after apply)
      + id                           = (known after apply)
      + instance_state               = (known after apply)
      + instance_type                = "t2.micro"
      + ipv6_address_count           = (known after apply)
      + ipv6_addresses               = (known after apply)
      + key_name                     = (known after apply)
```

This output shows the execution plan, describing which actions Terraform will take in order to change real infrastructure to match the configuration.

The output format is similar to the diff format generated by tools such as Git. The output has a + next to aws_instance.nemnous, meaning that Terraform will create this resource. Beneath that, it shows the attributes that will be set. When the value displayed is (known after apply), it means that the value won't be known until the resource is created.

Terraform will now pause and wait for your approval before proceeding. If anything in the plan seems incorrect or dangerous, it is safe to abort here with no changes made to your infrastructure.

In this case the plan is acceptable, so type yes at the confirmation prompt to proceed. Executing the plan will take a few minutes since Terraform waits for the EC2 instance to become available.

And to destroy the instance we can use **terraform destroy.**

Deploying Hello-world Spring Application in AWS

In this we will deploy a simple spring application in AWS Elastic Bean Stalk using terraform we'll also cover the variables in the process.
Each input variable accepted by a module must be declared using a variable block:

Variables:

Input variables serve as parameters for a Terraform module, allowing aspects of the module to be customized without altering the module's own source code, and allowing modules to be shared between different configurations.

Declaring a variable:

When you declare variables in the root module of your configuration, you can set their values using CLI options and environment variables. When you declare them in child modules, the calling module should pass values in the module block.

```
variable "api_dist" {
 description = "name of the disribution file"
 type        = string
 default     = "spring-boot-hello-world-rest-api.jar"
}
variable "environment" {
 type        = string
 default     = "test"
 description = "Environment, e.g. 'prod', 'staging', 'dev', 'pre-prod', 'UAT'"
}
```

The name of a variable can be any valid identifier except the following:

- source
- version
- providers
- count
- for_each
- lifecycle
- depends_on
- Locals

The variable declaration can optionally include a type argument to specify what value types are accepted for the variable, as described in the following section.

Prerequisites:

1. An Spring application - in this we have a simple Hello world spring application.
2. Building of a jar file of that spring application.

Plan of action:

1. Providers - we will be using AWS
2. We'll zip the application jar file using terraform.
3. Create an S3 bucket and a bucket object for the deployment in terraform. This is where your zip will be published when you run `terraform apply`.
4. Creating a Elastic bean stalk application
5. Creating Elastic Bean stalk environment

Providers

The provider block configures the named provider, in our case aws, which is responsible for creating and managing resources. A provider is a plugin that Terraform uses to translate the API interactions with the service. A provider is responsible for understanding API interactions and exposing resources. Because Terraform can interact with any API, you can represent almost any infrastructure type as a resource in Terraform.

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

Zipping the jar file:

Terraform's data.archive_file directive. It takes a file path and zips it for you. Point this to your image.
If you are using Docker images, look into ECR instead.

```
# create a zip of your deployment with terraform
data "archive_file" "api_dist_zip" {
 type        = "zip"
 source_file = "${path.root}/${var.api_dist}"
 output_path = "${path.root}/${var.api_dist}.zip"
}
```

Creating an S3 bucket :

create an S3 bucket and a bucket object for the deployment in terraform.
This is where your zip will be published when you run `terraform apply`.

```
resource "aws_s3_bucket" "dist_bucket" {
 bucket = "nemnous-elb-dist"
 acl    = "private"
}

resource "aws_s3_bucket_object" "dist_item" {
 key    = "s3_bucket_test_key"
 bucket = aws_s3_bucket.dist_bucket.id
 source = var.dist_zip
}
```

Creating an Elastic bean stalk application:

We'll configure the elastic bean stalk application for this we need to a service role which suits for initializing this EBS application.

```
resource "aws_elastic_beanstalk_application" "ebs-app" {
 depends_on = [aws_s3_bucket_object.dist_item]
 name = "spring-helloworld"
 description = "sample spring-helloworld elastic environment"
 appversion_lifecycle {
  service_role = "arn:aws:iam::736271802724:role/aws-service-
role/elasticbeanstalk.amazonaws.com/AWSServiceRoleForElasticBeanstalk"
  max_count = 128
  delete_source_from_s3 = true
 }
}

resource "aws_elastic_beanstalk_application_version" "ebs-app-ver" {
 depends_on = [aws_elastic_beanstalk_application.ebs-app]
 application = aws_elastic_beanstalk_application.ebs-app.name
 bucket = aws_s3_bucket_object.dist_item.bucket
 key = "s3_bucket_test_key"
 name = "v1"
}
```

Elastic Bean Stalk environment:

Now we'll create an Elastic bean stalk environment:

```
resource "aws_elastic_beanstalk_environment" "ebs-env" {
 depends_on = [aws_elastic_beanstalk_application_version.ebs-app-ver]
 name = "sample-application-dev"
 application = aws_elastic_beanstalk_application.ebs-app.name
 solution_stack_name = "64bit Amazon Linux 2 v3.0.3 running Corretto 11"
 cname_prefix = "nemnous"
 version_label = aws_elastic_beanstalk_application_version.ebs-app-ver.name

 setting {
    namespace = "aws:autoscaling:launchconfiguration"
    name      = "IamInstanceProfile"
    value     = "aws-elasticbeanstalk-ec2-role"
  }

}
```

And we are done with the configuration part for deploying these resources we will do **terraform init** and **terraform apply**.

Deploying Hello-world Spring Application in AWS VPC

In this we'll deploy the previously built Spring application within AWS - VPC.

Let's list out all resources to be created for running EC2 instance inside a virtual private network as that helps us in better understanding.

1. VPC
2. Subnet inside VPC
3. Internet gateway associated with VPC
4. Route Table inside VPC with a route that directs internet-bound traffic to the internet gateway
5. Route table association with our subnet to make it a public subnet.

We are going to create all these resources with Terraform.
As we have a clear picture of what resources we need to create for our production environment with EC2, we can try and identify some of our inputs as variables so that they are easily changed and maintained.

1. CIDR block for VPC
2. CIDR block for subnet which is a subset of CIDR block of VPC
3. Availability zone which is used to create our subnet

As always let's set provider for terraform as AWS with region

```
#providers
provider "aws" {
 region = var.region
}
```

Once the provider is set, let's add all variables which we are going to use while creating our resources.

```
# Variables
# CIDR block for VPC
# CIDR block for subnet which is a subset of CIDR block of VPC
# Availability zone which is used to create our subnet
# Instance type of EC2 instance
# With AWS, we can start adding tags to track our resources. We are using environment tag.
variable "region" {
 default = "us-east-1"
}
variable "cidr_vpc" {
 description = "CIDR block for the VPC"
 default = "10.1.0.0/16"
}
variable "cidr_subnet" {
 description = "CIDR block for the subnet"
 default = "10.1.0.0/24"
}
variable "availability_zone" {
 description = "availability zone to create subnet"
 default = "us-east-1a"
}
variable "instance_ami" {
  description = "AMI for aws EC2 instance"
  default = "ami-0cf31d971a3ca20d6"
}
variable "instance_type" {
  description = "type for aws EC2 instance"
  default = "t2.micro"
}
variable "environment_tag" {
  description = "Environment tag"
  default = "Learning"
}
variable "api_dist" {
  description = "name of the disribution file"
  type        = string
  default     = "spring-boot-hello-world-rest-api.jar"
}
variable "environment" {
  type        = string
  default     = "test"
  description = "Environment, e.g. 'prod', 'staging', 'dev', 'pre-prod', 'UAT'"
}
variable "namespace" {
  type        = string
  description = "Namespace, which could be your organization name, e.g. 'eg' or 'cp'"
  default     = "nemnous"
}

variable "dist_zip" {
  type        = string
  default     = "spring-boot-hello-world-rest-api.jar.zip"
}
```

As we have defined our variables, let us use them while creating our resources one by one. We are going to create VPC with defined CIDR block, enable DNS support and DNS hostnames so each instance can have a DNS name along with IP address.

```
resource "aws_vpc" "vpc" {
  cidr_block = var.cidr_vpc
  tags = {
    Environment = var.environment_tag
  }
}
```

Internet gateway needs to be added inside VPC which can be used by subnet to access the internet from inside.

```
resource "aws_internet_gateway" "igw" {
  vpc_id = aws_vpc.vpc.id
  tags = {
    Environment = var.environment_tag
  }
}
```

The subnet is added inside VPC with its own CIDR block which is a subset of VPC CIDR block inside given availability zone.

```
resource "aws_subnet" "subnet_public" {
 vpc_id = aws_vpc.vpc.id
 cidr_block = var.cidr_subnet
 map_public_ip_on_launch = "true"
 availability_zone = var.availability_zone
 tags = {
  Environment = var.environment_tag
 }
}
```

Route table needs to be added which uses internet gateway to access the internet.

```
resource "aws_route_table" "rtb_public" {
 vpc_id = aws_vpc.vpc.id
 route {
    cidr_block = "0.0.0.0/0"
    gateway_id = aws_internet_gateway.igw.id
 }
 tags = {
  Environment = var.environment_tag
 }
}
```

Once route table is created, we need to associate it with the subnet to make our subnet public.

```
resource "aws_route_table_association" "rta_subnet_public" {
 subnet_id      = aws_subnet.subnet_public.id
 route_table_id = aws_route_table.rtb_public.id
}
```

Once we have our networking setup ready, we will configure these VPC and subnet with the previous EBS application. We can do that by just adding the settings

```
resource "aws_elastic_beanstalk_environment" "ebs-env" {
 depends_on = [aws_elastic_beanstalk_application_version.ebs-app-ver]
 name = "sample-application-with-vpc"
 application = aws_elastic_beanstalk_application.ebs-app.name
 solution_stack_name = "64bit Amazon Linux 2 v3.0.3 running Corretto 11"
 cname_prefix = "nemnous"
 version_label = aws_elastic_beanstalk_application_version.ebs-app-ver.name

 setting {
    namespace = "aws:ec2:vpc"
    name = "VPCId"
    value = aws_vpc.vpc.id
  }
 setting {
    namespace = "aws:ec2:vpc"
    name = "AssociatePublicIpAddress"
    value = "true"
  }
 setting {
  namespace = "aws:ec2:vpc"
  name = "Subnets"
  value = aws_subnet.subnet_public.id
 }
```

```
  setting {
    namespace = "aws:autoscaling:launchconfiguration"
    name      = "IamInstanceProfile"
    value     = "aws-elasticbeanstalk-ec2-role"
  }

}
```

And once this is done we can run terraform init and terraform apply to deploy.