Building an AWS VPC with Terraform Step-by-Step

Published: 20 April 2021 - 7 min. read

TERRAFORM

Shanky

Read more tutorials by Shanky!

Table of Contents

If you need to set up AWS Virtual Private Cloud (VPC), you *could* make it happen via the AWS Management Console but automating it is so much more fun! In a DevOps scenario, building AWS services via tools like Terraform is a more scalable and automated approach to cloud resource provisioning.

In this tutorial, you will learn how to build and run a Terraform configuration to build a VPC with Terraform from scratch!

Prerequisites

This post will be a step-by-step tutorial. If you'd like to follow along, ensure you have the following in place:

- An AWS account
- Terraform – This tutorial will use Terraform v0.14.9 running on Ubuntu 18.04.5 LTS, but any operating system with Terraform should work.

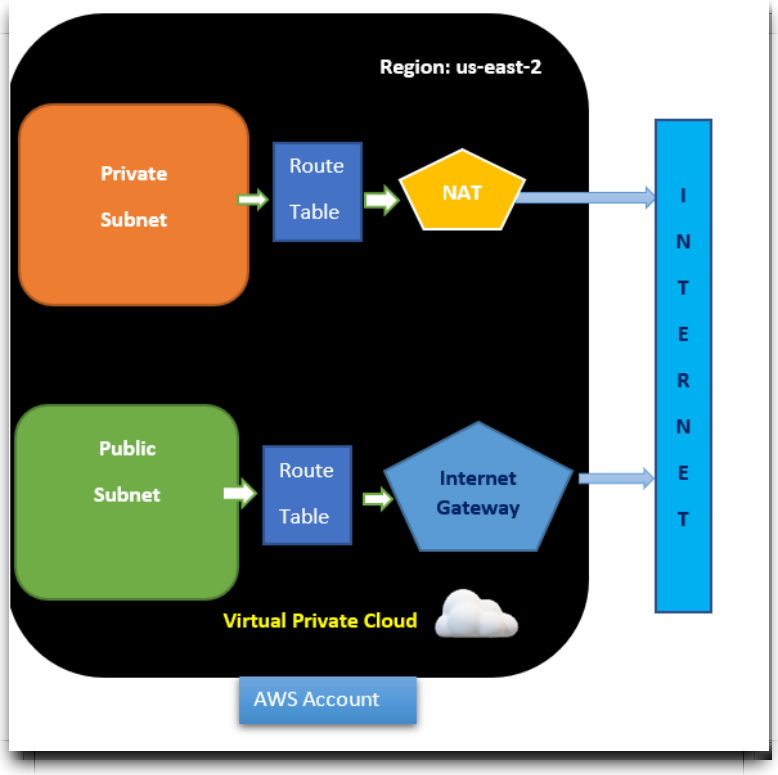
> **Related:** How to Install Terraform on Windows

- A code editor – Even though you can use any text editor to work with Terraform configuration files, you should have one that understands the HCL Terraform language. Try out Visual Studio (VS) Code.

Understanding AWS VPCs

The AWS cloud has dozens of various services from compute, storage, networking, and more. Each of these services can communicate with each other just like an on-prem datacenter's services. But, each of these services is independent of one another, and they are not necessarily isolated from other services. The AWS VPC changes that.

An AWS VPC is a single network that allows you to launch AWS services within a single isolated network. Technically, an AWS VPC is almost the same as owning a datacenter but with built-in additional benefits of scalability, fault-tolerance, unlimited storage, etc.



AWS VPC architecture

AWS VPCs are restricted by region. You cannot have one VPC spanning across regions with up to five VPCs supported per region.

Building the Terraform Configuration for an AWS VPC

Enough talk, let's get down to building!

1. To start, create a folder to store your Terraform configuration files in. This tutorial will create a folder called *terraform-vpc-demo* in your home directory.

```
mkdir ~/terraform-vpc-demo
 cd ~/terraform-vpc-demo
```

2. Open your favorite code editor and copy/paste the following configuration already created for you saving the file as *main.tf* inside of the *~/terraform-vpc-demo* directory. Information about each resource to create is inline.

The *main.tf* file contains all the resources which are required to be provisioned.

> **Terraform uses several different types of configuration files. Each file is written in either plain text format or JSON format. They have a specific naming convention of either .tf or .tfjson format.**

The Terraform configuration below:

- Creates a VPC
- Creates an Internet Gateway and attaches it to the VPC to allow traffic within the VPC to be reachable by the outside world.
- Creates a public and private subnet

> *Subnets are networks within networks. They are designed to help network traffic flow be more efficient and provide smaller, more manageable 'chunks' of IP addresses*

- Creates a route table for the public and private subnets and

associates the table with both subnets

- Creates a NAT Gateway to enable private subnets to reach out to the internet without needing an externally routable IP address assigned to each resource.

```
Create the VPC
 resource "aws_vpc" "Main" {                # Creating V
   cidr_block       = var.main_vpc_cidr     # Defining t
   instance_tenancy = "default"
 }
 Create Internet Gateway and attach it to VPC
 resource "aws_internet_gateway" "IGW" {    # Creating I
    vpc_id =  aws_vpc.Main.id               # vpc_id wil
 }
 Create a Public Subnets.
 resource "aws_subnet" "publicsubnets" {    # Creating P
   vpc_id =  aws_vpc.Main.id
   cidr_block = "${var.public_subnets}"        # CIDR bl
 }
 Create a Private Subnet                   # Creating Pr
 resource "aws_subnet" "privatesubnets" {
   vpc_id =  aws_vpc.Main.id
   cidr_block = "${var.private_subnets}"          # CIDR
 }
 Route table for Public Subnet's
 resource "aws_route_table" "PublicRT" {    # Creating R
    vpc_id =  aws_vpc.Main.id
         route {
    cidr_block = "0.0.0.0/0"               # Traffic fro
    gateway_id = aws_internet_gateway.IGW.id
     }
 }
 Route table for Private Subnet's
 resource "aws_route_table" "PrivateRT" {    # Creating
   vpc_id = aws_vpc.Main.id
   route {
   cidr_block = "0.0.0.0/0"             # Traffic from P
   nat_gateway_id = aws_nat_gateway.NATgw.id
   }
 }
 Route table Association with Public Subnet's
 resource "aws_route_table_association" "PublicRTassocia
    subnet_id = aws_subnet.publicsubnets.id
    route_table_id = aws_route_table.PublicRT.id
 }
 Route table Association with Private Subnet's
 resource "aws_route_table_association" "PrivateRTassoci
    subnet_id = aws_subnet.privatesubnets.id
    route_table_id = aws_route_table.PrivateRT.id
 }
 resource "aws_eip" "nateIP" {
   vpc   = true
 }
 Creating the NAT Gateway using subnet_id and allocation
```

```
resource "aws_nat_gateway" "NATgw" {
  allocation_id = aws_eip.nateIP.id
  subnet_id = aws_subnet.publicsubnets.id
}
```

3. Now, create another file inside *~/terraform-vpc-demo* directory, name it *vars.tf* and paste the content below.

> Vars.tf is a Terraform variables file that contains all the variables that the configuration file references.

You can see variables references in the configuration file by:

- `var.region`
- `var.main_vpc_cidr`
- `var.public_subnets`
- `var.private_subnets`

```
variable "region" {}
 variable "main_vpc_cidr" {}
 variable "public_subnets" {}
 variable "private_subnets" {}
```

> It is possible to include all configuration values in a single configuration file. To keep things clear and make it easier for

> **developers and admins, it is important to break things up.**

4. Create one more file inside *~/terraform-vpc-demo* directory, paste in the following code, and name it as *provider.tf* to define the AWS provider. The tutorial will be creating resources in the `us-east-2` region.

> **The providers file defines providers such as AWS, Oracle or Azure, etc., so that Terraform can connect with the correct cloud services.** *`provider "aws" { region = "us-east-2" }`*

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

5. Finally, create one more file inside the *~/terraform-vpc-demo* directory, name it *terraform.tfvars,* and paste the code below. This variables file contains the values that Terraform will use to replace the variable references inside of the configuration file.

```
main_vpc_cidr = "10.0.0.0/24"
 public_subnets = "10.0.0.128/26"
 private_subnets = "10.0.0.192/26"
```

Now, open your favorite terminal (Bash in this case) and verify all of the required files below are contained in the folder by running the `tree` command.

```
tree terraform-vpc-demo
```

```
root@ip-10-111-4-53:/home/ubuntu# tree terraform-vpc-demo/
terraform-vpc-demo/
├── main.tf
├── provider.tf
├── terraform.tfvars
└── vars.tf

0 directories, 4 files
```

Folder structure of terraform files

Running Terraform to Create the AWS VPC

Now that you have the Terraform configuration file and variables files ready to go, it's time to initiate Terraform and create the VPC! To provision, a Terraform configuration, Terraform typically uses a three-stage approach terraform init → terraform plan → terraform apply. Let's walk through each stage now.

1. Open a terminal and navigate to the *~\terraform-vpc-demo* directory.

```
cd ~\terraform-vpc-demo
```

2. Run the `terraform init` command in the same directory. The `terraform init` command initializes the plugins and providers which are required to work with resources.

```
terraform init
```

If all goes well, you should see the message `Terraform has been successfully initialized` in the output, as shown below.

```
root@ip-10-111-4-53:/home/ubuntu/terraform-vpc-demo# terraform init

Initializing the backend...

Initializing provider plugins...
- Using previously-installed hashicorp/aws v3.35.0

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, we recommend adding version constraints in a required_providers block
in your configuration, with the constraint strings suggested below.

* hashicorp/aws: version = "~> 3.35.0"


Warning: Interpolation-only expressions are deprecated

  on main.tf line 18, in resource "aws_subnet" "publicsubnets":
  18:   cidr_block = "${var.public_subnets}"        # CIDR block of public subnets

Terraform 0.11 and earlier required all non-constant expressions to be
provided via interpolation syntax, but this pattern is now deprecated. To
silence this warning, remove the "${ sequence from the start and the }"
sequence from the end of this expression, leaving just the inner expression.

Template interpolation syntax is still used to construct strings from
expressions when the template includes multiple interpolation sequences or a
mixture of literal strings and interpolations. This deprecation applies only
to templates that consist entirely of a single interpolation sequence.

(and one more similar warning elsewhere)

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 initialized successfully

3. Now, run the `terraform plan` command. This is an optional, yet recommended action to ensure your configuration's syntax is correct and gives you an overview of which resources will be provisioned in your infrastructure. `terraform plan`

```
terraform plan
```

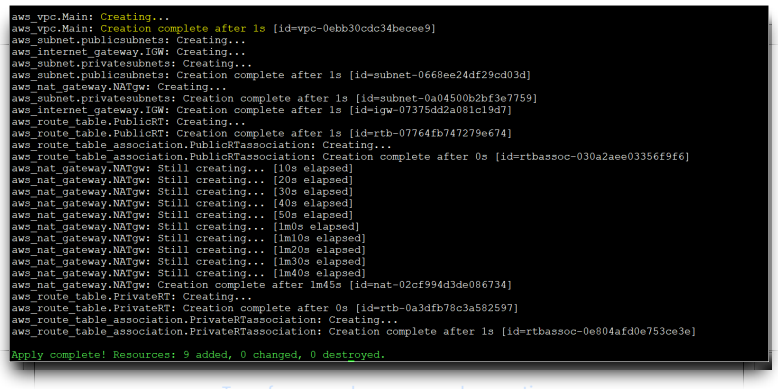
If successful, you should see a message like `Plan: "X" to add, "Y" to change, or "Z" to destroy` in the output to indicate the command was successful. You will also see every AWS resource Terraform intends to create.

```
  # aws_subnet.publicsubnets will be created
  + resource "aws_subnet" "publicsubnets" {
      + arn                                            = (known after apply)
      + assign_ipv6_address_on_creation = false
      + availability_zone                        = (known after apply)
      + availability_zone_id                    = (known after apply)
      + cidr_block                                   = "10.0.0.128/26"
      + id                                               = (known after apply)
      + ipv6_cidr_block_association_id  = (known after apply)
      + map_public_ip_on_launch          = false
      + owner_id                                     = (known after apply)
      + tags_all                                     = (known after apply)
      + vpc_id                                       = (known after apply)
    }

  # aws_vpc.Main will be created
  + resource "aws_vpc" "Main" {
      + arn                                  = (known after apply)
      + assign_generated_ipv6_cidr_block = false
      + cidr_block                           = "10.0.0.0/24"
      + default_network_acl_id           = (known after apply)
      + default_route_table_id           = (known after apply)
      + default_security_group_id        = (known after apply)
      + dhcp_options_id                  = (known after apply)
      + enable_classiclink               = (known after apply)
      + enable_classiclink_dns_support   = (known after apply)
      + enable_dns_hostnames             = (known after apply)
      + enable_dns_support               = true
      + id                               = (known after apply)
      + instance_tenancy                 = "default"
      + ipv6_association_id              = (known after apply)
      + ipv6_cidr_block                  = (known after apply)
      + main_route_table_id              = (known after apply)
      + owner_id                         = (known after apply)
      + tags_all                         = (known after apply)
    }

Plan: 10 to add, 0 to change, 0 to destroy.
```

Plan command execution

4. Next, tell Terraform actually to provision the AWS VPC and resources using `terraform apply`. When you invoke `terraform apply`, Terraform will read the configuration (*main.tf*) and the other files to compile a configuration. It will then send that configuration up to AWS as instructions to build the VPC and other components.

```
terraform apply
```



Terraform apply command execution

> **Notice the IDs defined in the Terraform output. You'll need these IDs to correlate the resources created in the next section.**

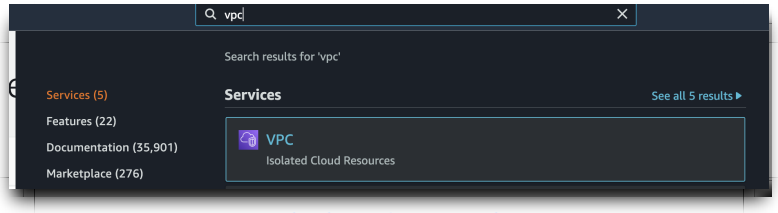
The Terraform command executed successfully, so let's switch over to the AWS Management Console to confirm that the VPC and components were created successfully.

Verifying the AWS VPC

By now, you should have created the VPC with Terraform. Let's verify by manually checking for the VPC in the AWS Management Console.

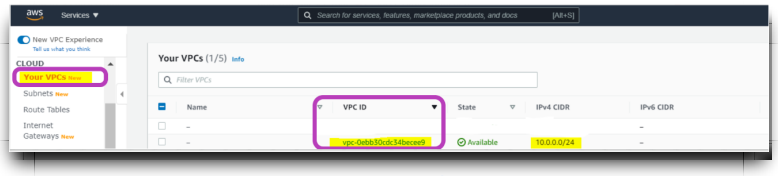
1. Open your favorite web browser and navigate to the AWS Management Console and log in.

2. While in the Console, click on the search bar at the top, search for 'vpc', and click on the **VPC** menu item.



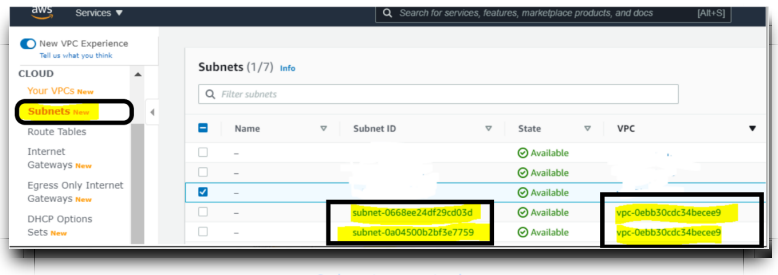
Navigating to the VPC service

3. Once on the VPC page, click on **Your VPCs**. You should see the VPC created with the same ID Terraform returned earlier.

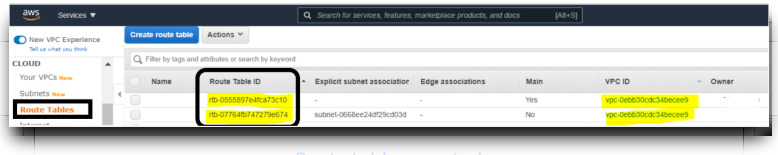


VPC created

4. Since Terraform created more than just the VPC resource but all of the resources required for the VPC, you should then find each resource on this page also.

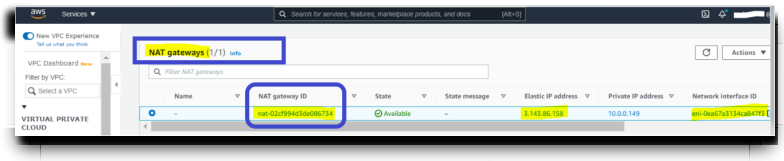


Subnets created



Route tables created

Internet Gateway created



NAT Gateway created

With all the components successfully created, using Terraform, this VPC is ready to go!

Conclusion

In this tutorial, you've learned how to use Terraform to deploy an AWS VPC and its components using Terraform.

Building an Amazon Virtual Private Cloud with Terraform allows you to create resources quickly, easily, and predictably. You are now ready to use this knowledge with other AWS services and build powerful services on top of it.

More from ATA Learning & Friends