

Only you can see this message X

This story's distribution setting is on. You're in the Partner Program, so this story is eligible to earn money. [Learn more](#)

Infrastructure as code - from Terraform to AWS

Terraform seems to be the go to tool these days for anything infrastructure as code, but is it possible to become an expert quickly? — by [Craig Godden-Payne@beardy.digital](#)



Craig Godden-Payne
May 2 · 6 min read ★

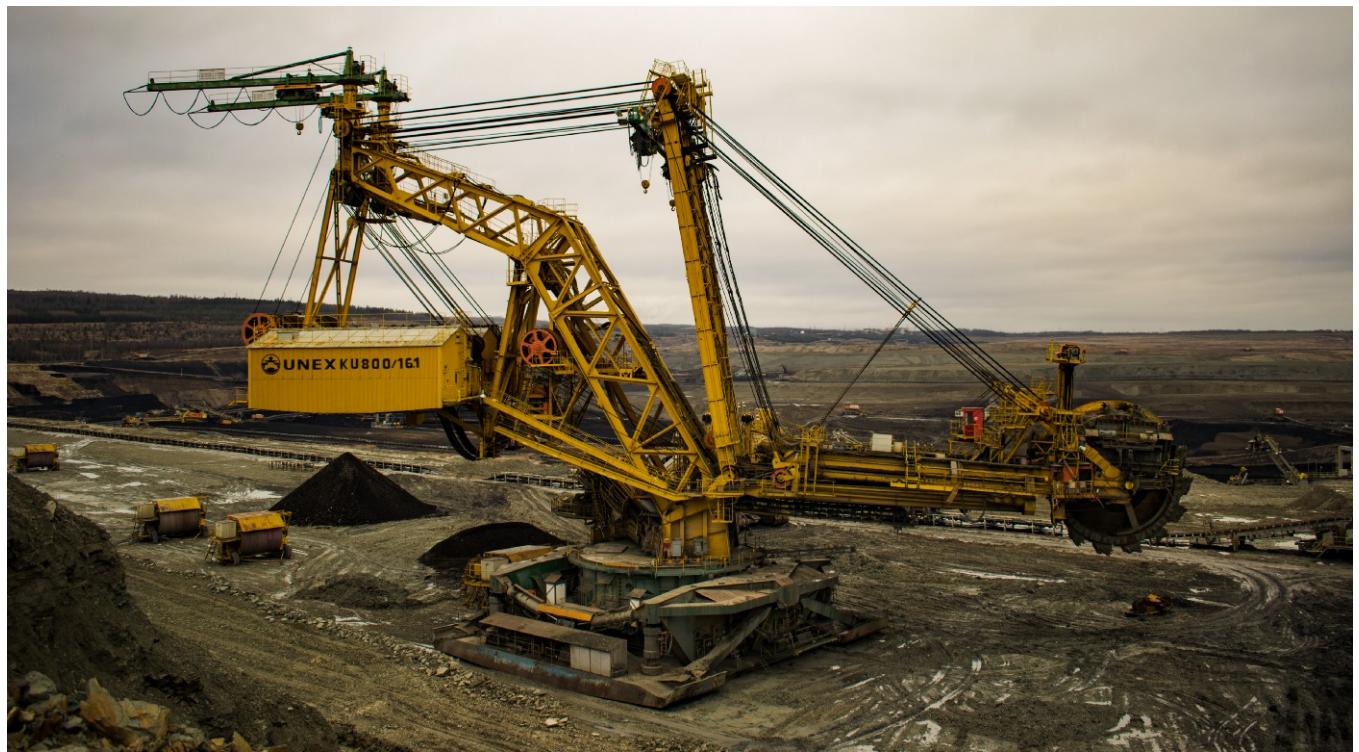


Photo by Michal Pech on Unsplash

Terraform enables you to safely and predictably create, change, and improve infrastructure. It is an open source tool that codifies APIs into declarative configuration files that can be shared amongst team members, treated as code, edited, reviewed, and versioned

If you've worked with CloudFormation before, it is a similar concept as in it provisions infrastructure, but it has much greater power, as it can be run against multiple cloud providers, such as Amazon Web Services, Microsoft Azure and Google Cloud Platform.



Photo by Pero Kalimero on Unsplash

I find the best way to learn a new tool, or to improve is to get some hands on experience.

This post hopes to guide you through a basic scenario, of setting up some infrastructure using terraform in the AWS cloud platform.

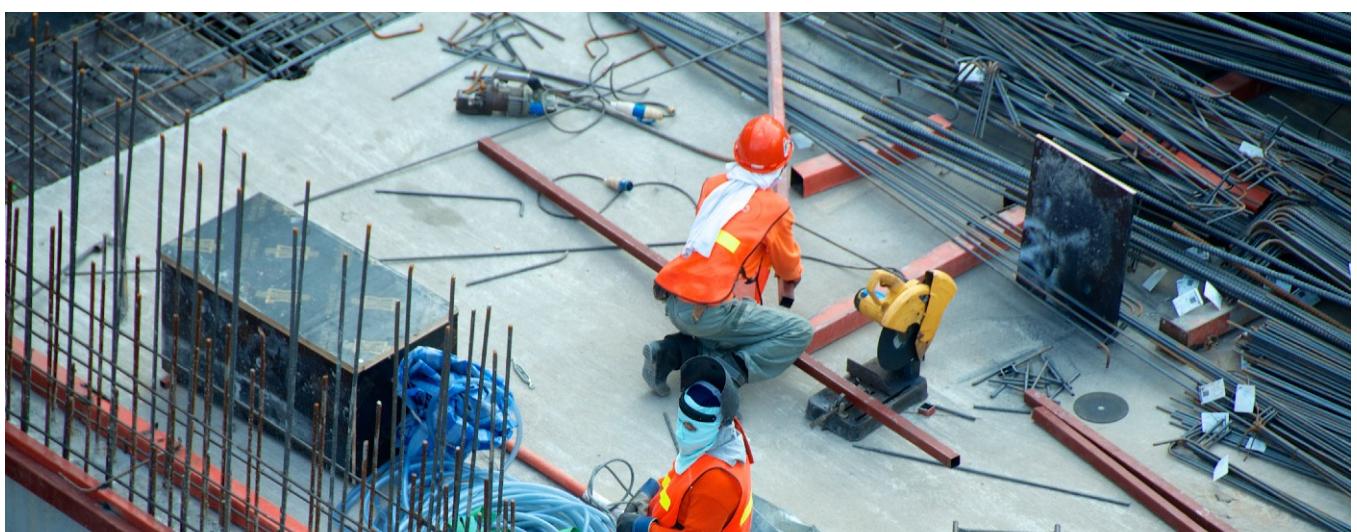




Photo by Etienne Girardet on Unsplash

• • •

Prerequisites

Download terraform. It's pretty simple on any operating system, and usually just involves downloading the binaries and adding them to the path of your shell.

Download Terraform - Terraform by HashiCorp

Download Terraform

- Terraform by HashiCorp Download Terraform www.terraform.io





Photo by Guilherme Cunha on Unsplash

• • •

Creating a user in AWS

In order to use terraform to provision infrastructure within AWS, you will need a way for terraform to authenticate with AWS. To do this, we are going to create a user in AWS, and use access key id and secret access key to authenticate

- Create a user, give it a sensible name (such as build-user) and make it an admin. In a real life scenario, it would be better to give this user the least amount of privilege, but for a demo, admin is fine.
- Generate some credentials, you will use these credentials for authentication.

Add user

1
2
3
4
5

Set user details

You can add multiple users at once with the same access type and permissions. [Learn more](#)

User name*	<input type="text" value="build-user"/>
------------	---

[+ Add another user](#)

Select AWS access type

Select how these users will access AWS. Access keys and autogenerated passwords are provided in the last step. [Learn more](#)

Access type*

Programmatic access

Enables an **access key ID** and **secret access key** for the AWS API, CLI, SDK, and other development tools.

AWS Management Console access

Enables a **password** that allows users to sign-in to the AWS Management Console.

[Create a build user](#)

Add user

1
2
3
4
5

▼ Set permissions

Add user to group	Copy permissions from existing user	Attach existing policies directly
-------------------	-------------------------------------	-----------------------------------

[Create policy](#)



Filter policies ▾ Search Showing 526 results

	Policy name ▾	Type	Used as
<input checked="" type="checkbox"/>	AdministratorAccess	Job function	Permissions policy (1)
<input type="checkbox"/>	AlexaForBusinessDeviceSetup	AWS managed	None
<input type="checkbox"/>	AlexaForBusinessFullAccess	AWS managed	None
<input type="checkbox"/>	AlexaForBusinessGatewayExecution	AWS managed	None
<input type="checkbox"/>	AlexaForBusinessPolyDelegatedAccessPolicy	AWS managed	None
<input type="checkbox"/>	AlexaForBusinessReadOnlyAccess	AWS managed	None
<input type="checkbox"/>	AmazonAPIGatewayAdministrator	AWS managed	None
<input type="checkbox"/>	AmazonAPIGatewayInvokeFullAccess	AWS managed	None

[Cancel](#)[Previous](#)[Next: Tags](#)

Add Administrator Access

Add user

[1](#)[2](#)[3](#)[4](#)[5](#)

Success

You successfully created the users shown below. You can view and download user security credentials. You can also email users instructions for signing in to the AWS Management Console. This is the last time these credentials will be available to download. However, you can create new credentials at any time.

Users with AWS Management Console access can sign-in at: <https://beardy-digital.signin.aws.amazon.com/console>

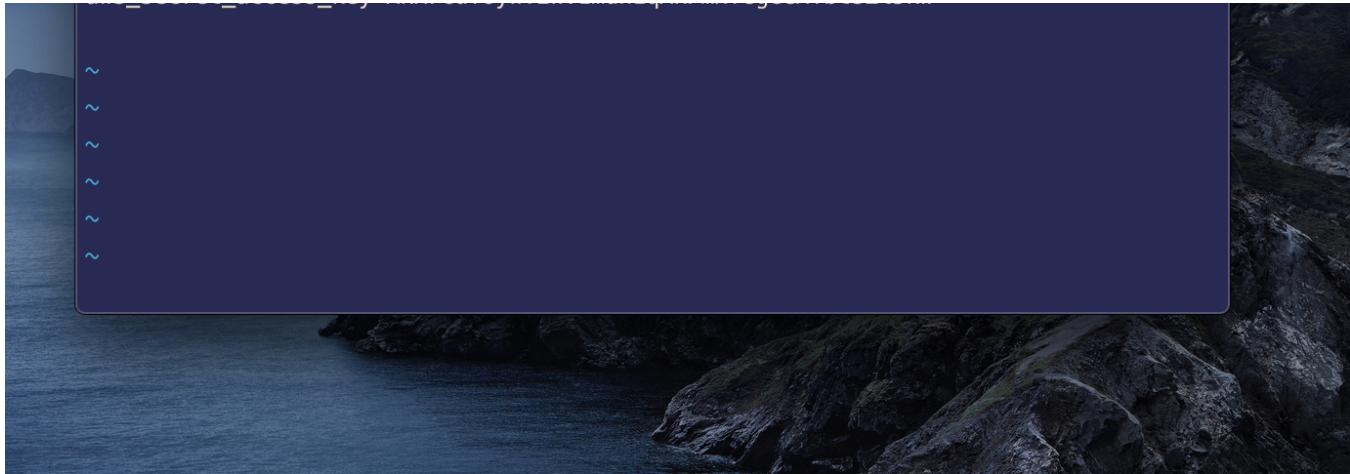
[Download .csv](#)

	User	Access key ID	Secret access key
<input checked="" type="checkbox"/>	build-user	AKIAX7VLMYDF4J5JPBET	AkR7cuYUyWT2Wv1mdh1q XkPmHYegGeMTbtsZl5kw Hide

Generating credentials

- Create and open a file located at `~/.aws/credentials` and add the required credentials

```
[build-user]
aws_access_key_id=AKIAIOSFODNN7EXAMPLE
aws_secret_access_key=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
```



- In order to use these credentials with terraform, make sure to set the AWS_PROFILE environment to the correct profile. You can do this like so:
- AWS_PROFILE=build-user terraform plan



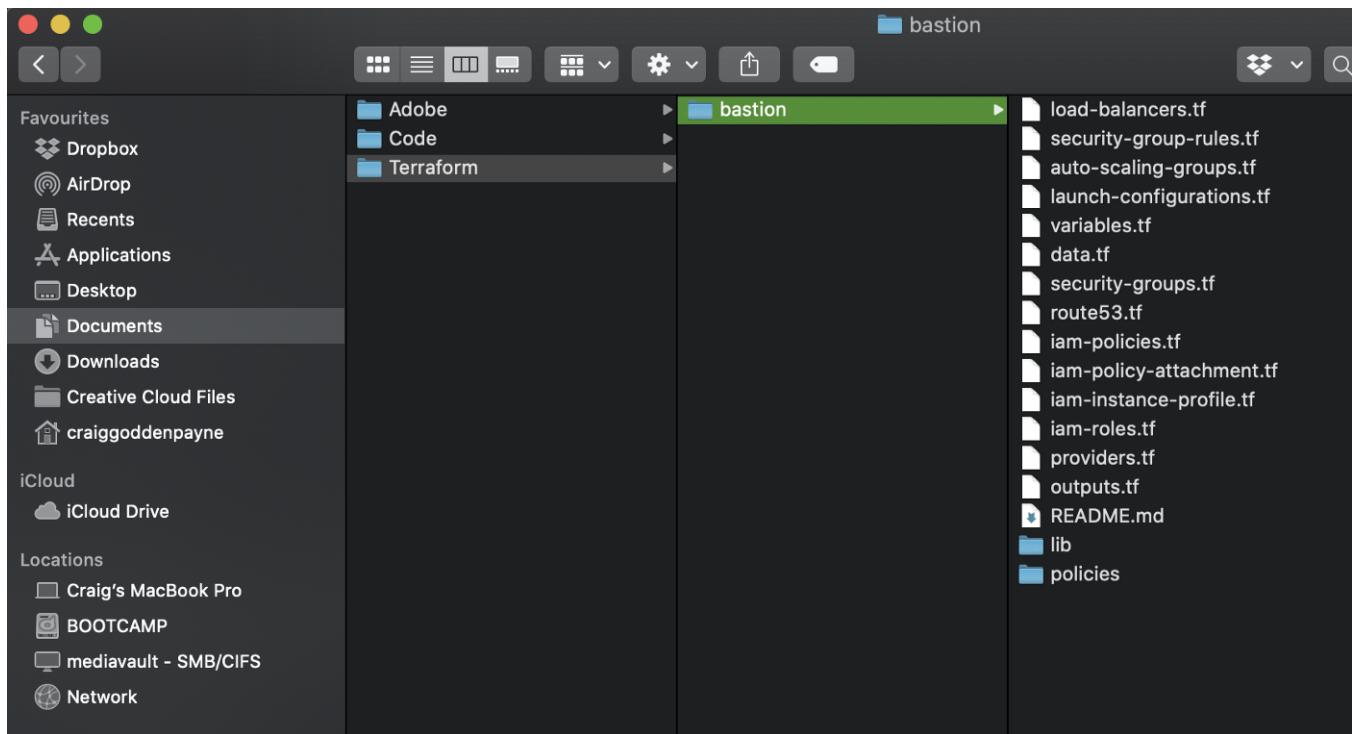
Photo by Jonny Caspari on Unsplash

• • •

Working with Terraform

Terraform is not fussy with project layout, it will interpret any files within the directory it uses, so it is up to you how you setup your project.

What I find is good is grouping infrastructure over multiple files, logically by the type of infra. There are no hard or fast rules about this, but I try to use common sense. It is not uncommon for me to have a project like this:



. . .

Working with State

One important feature to understand about terraform is the terraform state. Without a state, terraform has a hard job of tracking what changes between deployments.

There are different backends for storing the state, but it's common to use something like S3 as it is relatively cheap, and can have versioning enabled, so you are able to look back at previous versions if you need to.

You can also easily add dynamo db as a backing store for a lock table, which has the benefits of not allowing multiple users running the same terraform deployment at the same time.

```
terraform {  
  backend "s3" {  
    bucket      = "my-terraform-state-bucket"  
    key         = "my-application/state/terraform.tfstate"  
    region      = "eu-west-2"  
    dynamodb_table = "my-terraform-lock-table"  
    encrypt     = true  
  }  
}
```

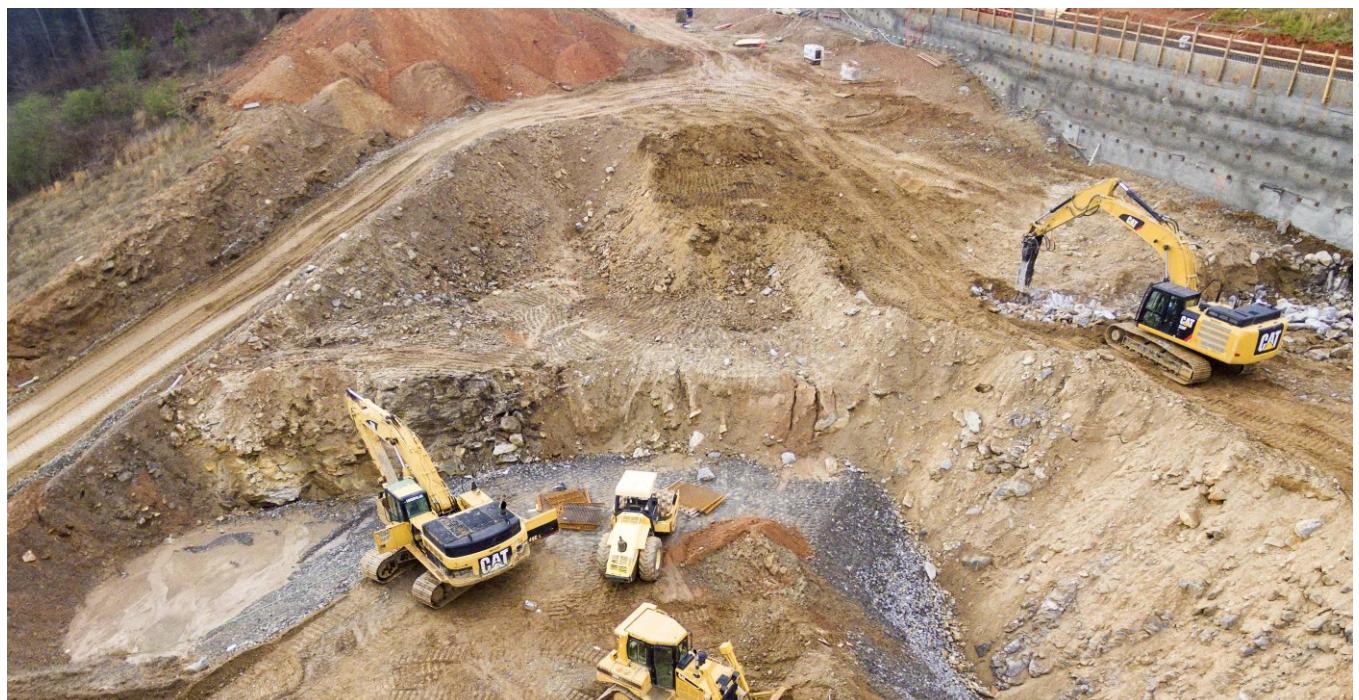




Photo by Shane McLendon on Unsplash

Lets set something simple up

As mentioned above, let's build something in terraform. Now we have the user created to be able to authenticate with AWS we will start with building a basic network.

I tend to use Jetbrains IDEs as I find the terraform support probably one of the best out of the IDEs I have used.

Create a file, lets call it providers.tf and let terraform know which region you want to deploy to:

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

Now create another file, let's call it vpc.tf, and inside let's create a VPC.

```
resource "aws_vpc" "vpc" {  
    cidr_block      = "10.0.0.0/16"  
    enable_dns_support = "true"  
    enable_dns_hostnames = "true"  
}
```

This example will create a VPC with IP range of 10.0.0.0 — 10.0.255.255, a total of 65536 (hopefully enough!!)

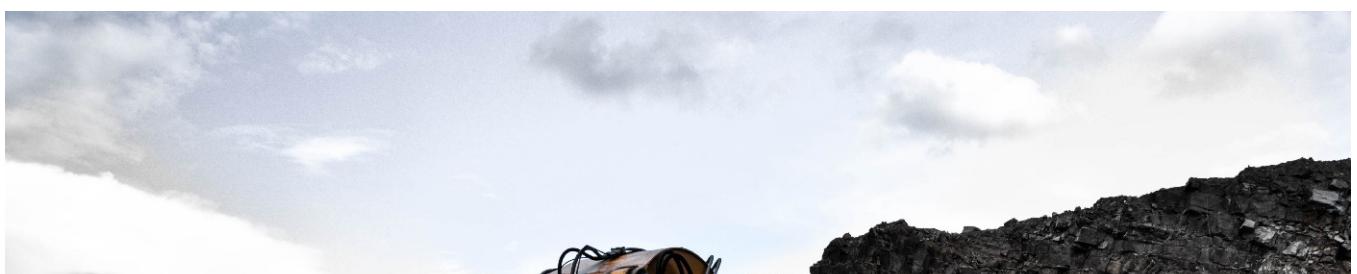




Photo by Jonny Caspari on Unsplash

• • •

Let's see what will be deployed

So now we have the credentials called build-user setup and terraform installed, all we need to do is in the console, navigate to the code directory with the vpc.tf code.

Run the command

```
AWS_PROFILE=build-user terraform init
```

```
AWS_PROFILE=build-user terraform plan
```

```
craiggoddenpayne@Craigs-MacBook-Pro: ~/Dropbox/BeardyDigital/Blogs
craiggoddenpayne@Craigs-MacBook-Pro ~ cd Dropbox/BeardyDigital/Blogs
craiggoddenpayne@Craigs-MacBook-Pro ~/Dropbox/BeardyDigital/Blogs > clear
craiggoddenpayne@Craigs-MacBook-Pro ~/Dropbox/BeardyDigital/Blogs > AWS_PROFILE=build-user terraform init

Initializing the backend...

Initializing provider plugins...
- Checking for available provider plugins...
- Downloading plugin for provider "aws" (hashicorp/aws) 2.60.0...

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, it is recommended to add version = "..." constraints to the
corresponding provider blocks in configuration, with the constraint strings
suggested below.

* provider.aws: version = "> 2.60"
```

```

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.
craiggoddenpayne@Craigs-MacBook-Pro ~$ ~/Dropbox/BeardyDigital/Blogs

```

terraform init

```

craiggoddenpayne@Craigs-MacBook-Pro: ~/Dropbox/BeardyDigital/Blogs
craiggoddenpayne@Craigs-MacBook-Pro ~$ ~/Dropbox/BeardyDigital/Blogs
AWS_PROFILE=build-user 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.

-----
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_vpc.vpc will be created
+ resource "aws_vpc" "vpc" {
    + arn
    + assign_generated_ipv6_cidr_block = false
    + cidr_block
    + default_network_acl_id
    + default_route_table_id
    + default_security_group_id
    + dhcp_options_id
    + enable_classiclink
    + enable_classiclink_dns_support
    + enable_dns_hostnames
    + enable_dns_support
    + id
    + instance_tenancy
    + ipv6_association_id
    + ipv6_cidr_block
    + main_route_table_id
    + owner_id
}

Plan: 1 to add, 0 to change, 0 to destroy.

-----
Note: You didn't specify an "-out" parameter to save this plan, so Terraform
can't guarantee that exactly these actions will be performed if
"terraform apply" is subsequently run.

```

terraform plan

Cool, so looks like a VPC will be created when we apply, but without setting up a remote state, we will get in a world of pain, as terraform will not know the current state of what is in AWS.

We can setup the S3 state as mentioned earlier.

Setting up the state

In the console, create an S3 bucket for storing terraform state files

Amazon S3 > Create bucket

Create bucket

General configuration

Bucket name

Bucket name must be unique and must not contain spaces or uppercase letters. [See rules for bucket naming](#)

Region

EU (London) eu-west-2

Create an S3 bucket

Add a file called `terraform.tf` and populate the content with the following, substituting your own bucket name:

```
terraform {
  backend "s3" {
    bucket = "craig-godden-payne-terraform-state"
    key    = "development-vpc/terraform.tfstate"
    region = "eu-west-2"
    acl    = "bucket-owner-full-control"
  }
}
```

Project ~/Dropbox/BeardyDigital/Blogs

- Blogs
- .terraform
- provider.tf
- terraform.tf**
- vpc.tf
- External Libraries
- Scratches and Consoles

```
1   terraform {
2     backend "s3" {
3       bucket = "craig-godden-payne-terraform-state"
4       key    = "development-vpc/terraform.tfstate"
5       region = "eu-west-2"
6       acl    = "bucket-owner-full-control"
7     }
8   }
```

• • •

Awesome, lets deploy it

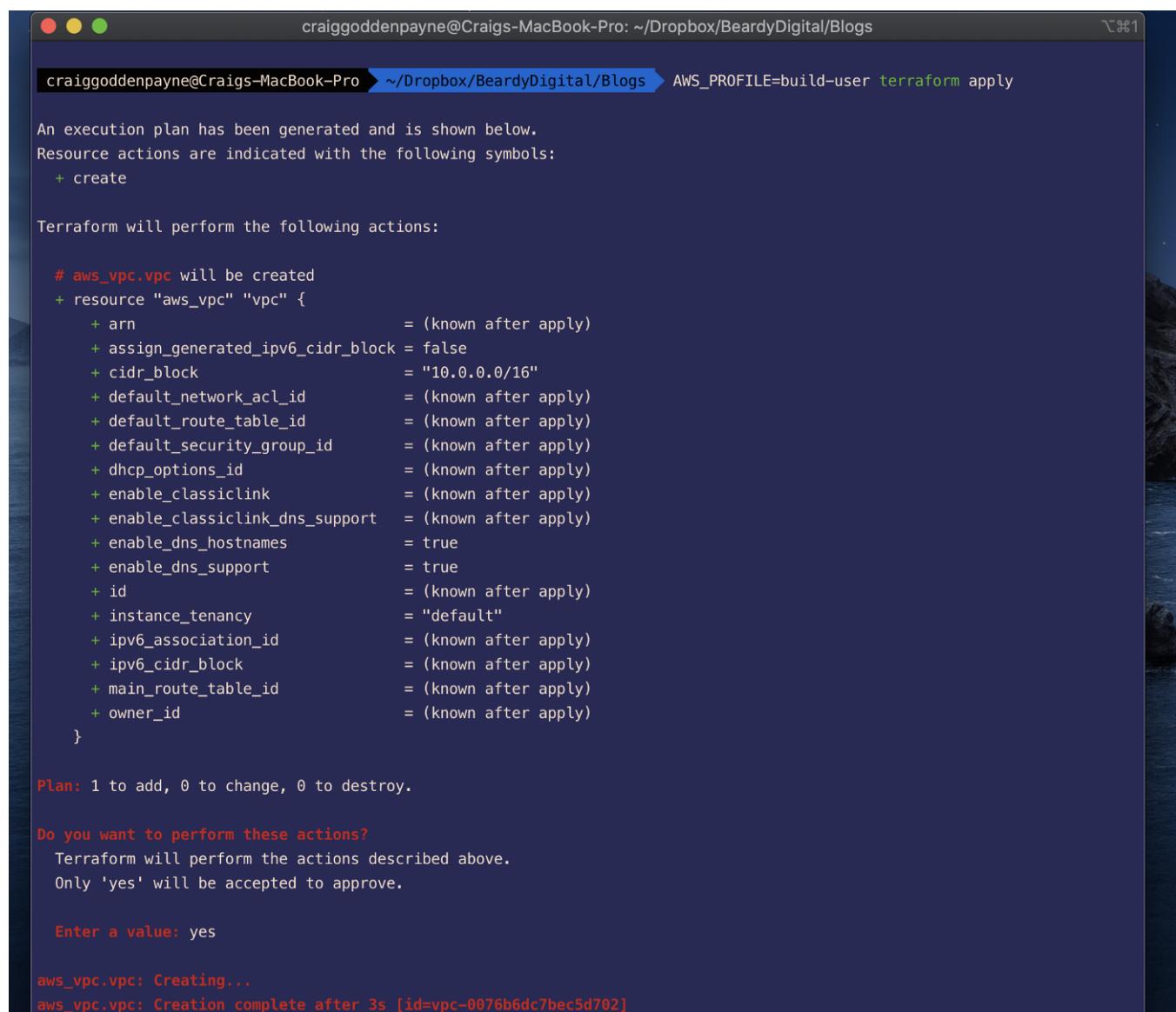
Now we have the state managed in S3, run the plan again to make sure that everything is as expected

Now, lets build some infra!

Run the command:

```
AWS_PROFILE=build-user terraform apply
```

Before the infra is deployed, terraform will ask you if you want to make changes, so make sure to answer `yes` to this question.



The screenshot shows a terminal window on a Mac OS X desktop. The title bar reads "craiggoddenpayne@Craigs-MacBook-Pro: ~/Dropbox/BeardyDigital/Blogs". The terminal content is as follows:

```
craiggoddenpayne@Craigs-MacBook-Pro: ~/Dropbox/BeardyDigital/Blogs ➤ AWS_PROFILE=build-user 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_vpc.vpc will be created
+ resource "aws_vpc" "vpc" {
    + arn                      = (known after apply)
    + assign_generated_ipv6_cidr_block = false
    + cidr_block                = "10.0.0.0/16"
    + 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       = true
    + 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)
}

Plan: 1 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

aws_vpc.vpc: Creating...
aws_vpc.vpc: Creation complete after 3s [id=vpc-0076b6dc7bec5d702]
```

```
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
craiggoddenpayne@Craigs-MacBook-Pro ~/Dropbox/BeardyDigital/Blogs
```

You just deployed your first VPC

Have a look in the console:

The screenshot shows the AWS VPC service page. At the top, there are buttons for 'Create VPC' and 'Actions'. A search bar is present above a table. The table has columns: Name, VPC ID, State, IPv4 CIDR, IPv6 CIDR, DHCP options set, and Main Route table. Two VPCs are listed:

Name	VPC ID	State	IPv4 CIDR	IPv6 CIDR	DHCP options set	Main Route table
vpc-0076b6dc7bec5d702	vpc-0076b6dc7bec5d702	available	10.0.0.0/16	-	dopt-0e2a7366	rtb-07df181f0bf3285c0
vpc-e475338c	vpc-e475338c	available	172.31.0....	-	dopt-0e2a7366	rtb-82b380ea

Below the table, a specific VPC is selected: 'vpc-0076b6dc7bec5d702'. The 'Description' tab is active, showing the following details:

- VPC ID: vpc-0076b6dc7bec5d702
- State: available
- IPv4 CIDR: 10.0.0.0/16
- IPv6 Pool: -
- Network ACL: acl-0c24dc494ed7adbbf
- DHCP options set: dopt-0e2a7366
- Owner: 549041520843
- Tenancy: default
- Default VPC: No
- IPv6 CIDR: -
- DNS resolution: Enabled
- DNS hostnames: Enabled
- Route table: rtb-07df181f0bf3285c0

The vpc we just created

Lets tear it back down

Deployment is easy, tearing down the infra is just as easy with the terraform destroy command.

To remove the VPC, use the following command

```
AWS_PROFILE=build-user terraform destroy
```

The terminal window shows the command being run:

```
craiggoddenpayne@Craigs-MacBook-Pro: ~/Dropbox/BeardyDigital/Blogs
craiggoddenpayne@Craigs-MacBook-Pro ~/Dropbox/BeardyDigital/Blogs AWS_PROFILE=build-user terraform destroy
aws_vpc.vpc: Refreshing state... [id=vpc-0076b6dc7bec5d702]
```

Output from Terraform:

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

Terraform will perform the following actions:
```

```
# aws_vpc.vpc will be destroyed
- resource "aws_vpc" "vpc" {
    - arn = "arn:aws:ec2:eu-west-2:549041520843:vpc/vpc-0076b6dc7bec5d702" -> null
    - assign_generated_ipv6_cidr_block = false -> null
    - cidr_block = "10.0.0.0/16" -> null
    - default_network_acl_id = "acl-0c24dc494ed7adbbf" -> null
    - default_route_table_id = "rtb-07df181f0bf3285c0" -> null
    - default_security_group_id = "sg-06246cdf22dcfef25" -> null
    - dhcp_options_id = "dopt-0e2a7366" -> null
    - enable_dns_hostnames = true -> null
    - enable_dns_support = true -> null
    - id = "vpc-0076b6dc7bec5d702" -> null
    - instance_tenancy = "default" -> null
    - main_route_table_id = "rtb-07df181f0bf3285c0" -> null
    - owner_id = "549041520843" -> null
    - tags = {} -> null
}

Plan: 0 to add, 0 to change, 1 to destroy.

Do you really want to destroy all resources?
Terraform will destroy all your managed infrastructure, as shown above.
There is no undo. Only 'yes' will be accepted to confirm.

Enter a value: yes

aws_vpc.vpc: Destroying... [id=vpc-0076b6dc7bec5d702]
aws_vpc.vpc: Destruction complete after 0s

Destroy complete! Resources: 1 destroyed.
craiggoddenpayne@Craigs-MacBook-Pro ~ ~/Dropbox/BeardyDigital/Blogs
```

terraform destroy

And there it does, all gone!

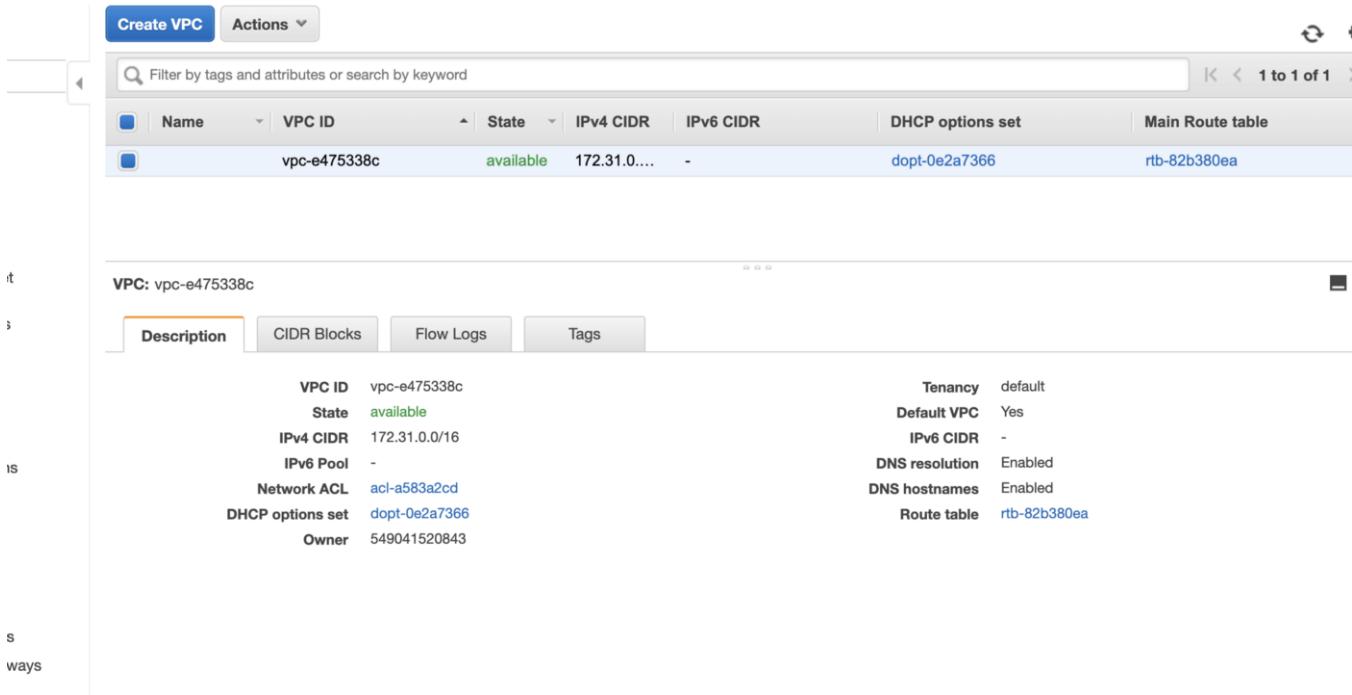




Photo by Markus Spiske on Unsplash

• • •

Conclusion

After reading this, you should be able to setup terraform, authenticate with AWS and be able to create infrastructure, manage its state and destroy infra within AWS.

To build on top of this example, just add more and more files containing AWS resources.

Terraform really is straightforward and the beauty is that you can repeat creating and tearing down infrastructure over and over, no longer you need to create things through the AWS console, and if you use in combination with source control, such as git, you can maintain a history of how the infrastructure has changed, and also who changed it!

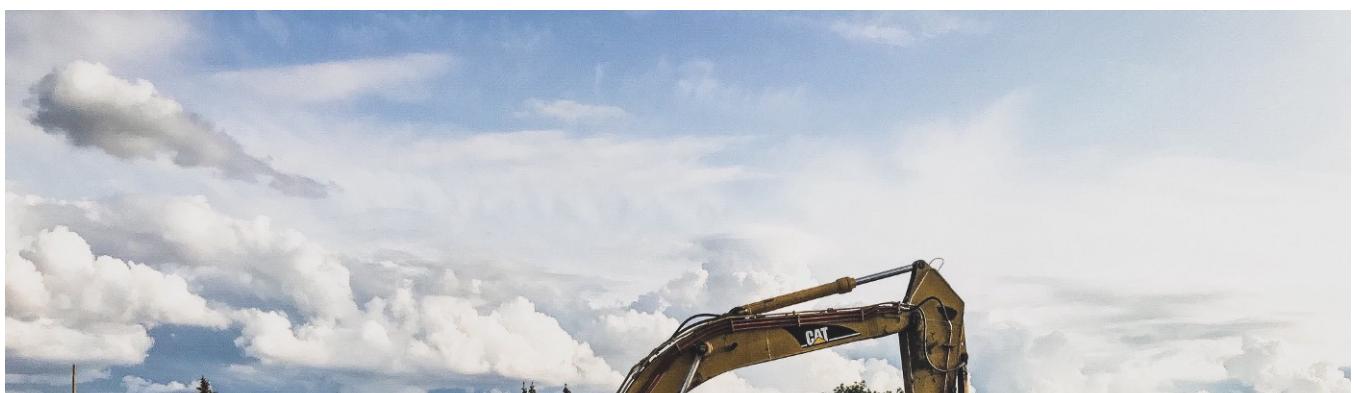




Photo by Jamar Penny on Unsplash

[AWS](#) [Terraform](#) [DevOps](#) [Development](#) [Infrastructure](#)

[About](#) [Help](#) [Legal](#)

Get the Medium app

