Blog About Contact   Zero to Devops

7 Oct

Ultimate Guide to Terraform: From Beginner to Expert

Below is my ultimate resource for learning Terraform via a written tutorial. It is everything that is needed to get anyone up and running on Terraform in a quick, easy manner and immediately jumps into the building of resources (the fun part).

Additionally it contains pretty much everything you would require as knowledge for the Hashicorp Terraform Associate Exam, so it doubles up as pretty good exam notes as well! Do note that while this guide is focussed on building AWS resources, the core concepts are the key takeaway and Terraform is designed to easily switch between providers (AWS, GCP etc.).

Lastly if you are looking for a good video course on Terraform I cannot recommend Zeal Vora's Udemy Course enough. In fact a lot of my notes from the course are included in this guide.

The Guide

Key
Purple - Phrases and Terms used specifically in Terraform and Terraform Code.
Blue- Highlights General Important Terms and Phrases.
Red - Command Line Arguments.

Blog About Contact   Zero to Devops

example an AMI on AWS is referred to as an AMI within Terraform as well.

Infrastructure as Code - Tools Overview

There are a lot of tools that allow you to deploy infrastructure as code:

- Terraform
- CloudFormation
- Heat
- Ansible
- SaltStack

Chef, Puppet, Others

Not all of these tools are targeted for the same purpose. It is important to understand the difference between Configuration Management and Infrastructure Orchestration:

- Ansible, Chef and Puppet are configuration management tools which means they are primarily designed to install and manage software on existing servers
- Terraform and CloudFormation are infrastructure orchestration tools which are designed to provision servers and infrastructure themselves.

You can use infrastructure orchestration tools and configuration management tools in tandem. For example you could use Terraform to create a new EC2 instance on AWS, Terraform can then call Ansible to install and configure software and applications on the EC2 instance.

Terraform Simple Benefits Overview

- Terraform supports multiple platforms - AWS, GCP, Azure etc.
- Terraform has a simple language and a fast learning curve. The syntax is easy to read and understand.
- Terraform is easy to integrate with configuration management tools like Ansible.
- It is easily extensible with plugins.
- It is free.

Creating a Simple EC2 Instance with Terraform

When launching a resource in AWS there are 3 things to consider:

- How will you authenticate to AWS?
- Which region will the resource be launched in?
- Which resource do you want to launch?

Blog About Contact   Zero to Devops

To use this you create a key pair for an AWS IAM User which Terraform will use to
authenticate to AWS.

It is important to note that while static credentials in your main file is an easy method
and useful as a tutorial or for hobby use it is not considered best practice.

First create a .tf file in this case I will use first_ec2.tf
Now you can edit the file to add Terraform code.

Example code for a simple EC2 instance:

```
provider "aws" {
    region = "us-west-2"
    access_key = "ExAmpLEKey21fjwljsfjkrf"
    secret_key = "EXamPLeSECreTKEySDAHhhhsJJn23Ksda"
}
resource "aws_instance" "my_ec2" {
    ami = "ami-05b622b5fa0269787"
    instance_type = "t2.micro"
}
```

As you can see in this example code we have included answers to the "3 important
things to consider". Lets run through the code step-by-step:

Statement 1 (defines what platform, where to launch and method of authentication)**:**
provider - In this case it is "aws", the provider has to be specified and it is simply the
platform that terraform will be provisioning infrastructure on.
   region - This is the AWS Region we want to launch our EC2 instance into. In this case
   we have chosen "us-west-2"
   access_key - This is the 'access key' from the key pair we created for our AWS IAM
   User, it is part of what allows Terraform to authenticate to AWS.
   secret_key - This is the 'secret key' from the key pair we created for our AWS IAM
   User.

Statement 2 (defines resource specfications)**:**
resource - In this case we have specified "aws_instance" the terraform name for an AWS
EC2 Instance and secondly "my_ec2" which is what we want to name the instance. Two
resource blocks cannot have the same name "my_ec2" will be a unique name in the file.
   ami - Specifies the Amazon Machine Image, ie. what image (OS specifics) we want to
   load on the EC2 Instance.
   instance_type - In this case we have specified "t2.micro" this specifies what type of
   Amazon EC2 we want our instance to be. (How much memory, storage etc.)

Every AWS resource has mandatory elements and optional elements to include in a
Terraform resource statement. For EC2 you need to specify the resource, ami and

Blog About Contact   Zero to Devops

included here as this is as simple as it gets for launching an EC2.

You can view the requirements for each AWS resource at the [Terraform Registry Documentation](#).

Now we have written our code and saved it with a .tf file type we can run the code with Terraform.

Step 1: Run "terraform init" on the command line in the folder containing your tf file - This will initialise Terraform and install any dependencies your file requires.

Step 2: Run "terraform plan" this will generate a plan of the infrastructure your file will create.

```
tclarke@DESKTOP━━━━━━:~/documents/learning/terraform$ terraform plan

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_instance.my_ec2 will be created
  + resource "aws_instance" "my_ec2" {
      + ami                          = "ami-05b622b5fa0269787"
      + 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"
```

Notice how the ami and instance_type are specified but a whole load of other options are "known after apply". These could have been specified but were optional. Bear in mind these options run a lot longer than this screenshot shows.

Step 3: If you are happy with the plan, run "terraform apply" to create the infrastructure.

Step 4: Terraform will show you the plan again and ask if you want to perform these actions. Say 'yes'.

Blog About Contact   Zero to Devops

```
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_instance.my_ec2: Creating...
aws_instance.my_ec2: Still creating... [10s elapsed]
aws_instance.my_ec2: Still creating... [20s elapsed]
aws_instance.my_ec2: Creation complete after 27s [id=i-0ede7e2ee93a01bb7]
```

Provided there are no errors during apply if you check your management console there
will now be an EC2 Instance running.

Providers

Terraform supports multiple providers:

Dependant on what type of infrastructure you want to launch you will have to use the
appropriate provider in Terraform.

This is important for the initialisation phase:

Whenever you add a provider it is important to run "`terraform init`" which will
download plugins associated with the provider.

```
tclarke@DESKTOP█████████:~/documents/learning/terraform$ terraform init

Initializing the backend...

Initializing provider plugins...
- Finding latest version of hashicorp/aws...
- Installing hashicorp/aws v3.33.0...
- Installed hashicorp/aws v3.33.0 (signed by HashiCorp)
```

This is from the EC2 example above, notice how Terraform has realised we are using the
AWS provider and has downloaded the associated AWS plugins.

In Terraform there are certain providers that are either "Hashicorp Maintained" (AWS,
Azure etc.) or "Non-Hashicorp Maintained". Below is best practice syntax for establishing
providers in your Terraform, notice how there are two blocks, one for what providers are
required and another for each provider specific config.

```
terraform {
  required_providers {
    aws = {
```

Blog About Contact   Zero to Devops

```
    ʃ
}

provider "aws" {
  # Configuration options
}
```

In Terraform v13+ this provider syntax is actually deemed best practice for all providers. You can get this code block ready made from the Terraform Registry website.

If you are comfortable in Terraform in one provider it is easy to switch between providers as the overall Terraform syntax is the same and only provider specific terms and resources change.

Resources

Resources are references to individual services a particular provider offers.
Eg. in AWS for an EC2 instance the Terraform resource is aws_instance

Resources in terraform can get very specific, below is a snapshot from the Terraform Registry of **some** AWS EC2 resource options:

aws_ec2_transit_gateway_route_table_
propagation

aws_ec2_transit_gateway_vpc_
attachment

aws_ec2_transit_gateway_vpc_
attachment_accepter

aws_eip

aws_eip_association

aws_instance

aws_key_pair

You can see aws_instance which we used in the example in this list. However there are so many more resource options which can be used.

Using a Non-AWS Provider (Example)

Blog About Contact   Zero to Devops

Below is the Terraform code to create a GitHub RepositoryCode:

```
terraform {
  required_providers {
    github = {
      source = "integrations/github"
      version = "4.6.0"
    }
  }
}

provider "github" {
    token = "b1_____"
}

resource "github_repository" "example" {
    name = "Terraform-Repo"
    visibility = "public"
}
```

You can see in the above we are using a separate code block to define the provider. As discussed this is actually now best practice and the simpler form used in the earlier EC2 AWS example is now slightly outdated.

The provider block can just be copied from the provider documentation on the TF Registry, click the "Use Provider" button to get the block with the above syntax. You can do this for any provider.

Note in the above example GitHub authentication is a "token" rather than access keys. The authentication method does change between providers and you can check what is needed in the TF Documentation. In this case all you have to do is request a token from your GitHub account in your browser and use it in Terraform.

To run this code and create a new GitHub repo we simply have to use:

- `terraform init` - This downloads the necessary packages for the GitHub provider
- `terraform plan` - Create a plan for what infrastructure will be changed
- `terraform apply` - Create the infrastructure

Destroying Resources With Terraform

`terraform destroy` - allows us to destroy all resources created within the folder.

Blog About Contact   Zero to Devops

Let's say I only want to destroy the EC2 instance.

In this case we can make use of a "target flag" to direct Terraform on what resources to destroy:

```
terraform destroy -target aws_instance.my_ec2
```

The target option can be used to focus Terraform's attention on only a subset of resources. You use a combination of Resource Type + Local Resource Name.

These are the Resource Types and LRNs we have created so far:

Resource Type	Local Resource Name
aws_instance	my_ec2
github_repository	example

Below is the code where Resource Type and Local Resource Name are defined:

```
resource "aws_instance" "my_ec2" {
    ami = "ami-05b622b5fa0269787"
    instance_type = "t2.micro"
}
```

```
resource "github_repository" "example" {
    name = "Terraform-Repo"
    visibility = "public"
}
```

- Resource Type - An unchangeable resource type used by Terraform that refers to a specific resource type for a provider
- Local Resource Name - The name you give to the resource you create, it is a custom value and can be anything you want. Do note this name only applies locally within your terraform code as a reference, it is not the name given to the resource itself.

So when using `terraform destroy -target aws_instance.my_ec2` - the "aws_instance.my_ec2" at the end tells Terraform exactly what resource to destroy. The syntax being "resource_type.local_resource_name".

When we run the destroy command with the target flag. Terraform will let us know it is specifically going to destroy only the resources we flagged:

```
tclarke@DESKTOP        :~/documents/learning/terraform$ terraform destroy -target aws_instance.my_ec2

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_instance.my_ec2 will be destroyed
```

Next we say "yes" please do destroy this.

Blog About Contact   Zero to Devops

longer be an instance running. It's that easy!

It is important to note that while we have destroyed this resource we have not removed the code to create the aws_instance from our first_ec2.tf config file.
Due to this if we now ran `terraform plan` again then Terraform will plan to create the aws_instance we just destroyed again.

If you don't want this to be the case then you can remove the code creating the aws_instance from the config file entirely or you could just comment it out.

Terraform State Files

Terraform stores the state of the infrastructure that are created in the .tf files in the Terraform State File.

This state file allows Terraform to map real world resources to your configuration files.

When we first ran `terraform init` on our initial first_ec2.tf file it created a state file within the folder.

Once we used `terraform apply` to bring up the EC2 instance Terraform then added the state of that EC2 Instance to the state file.

This is how Terraform knows what needs creating and what doesn't when subsequent changes are made in the folder. For example when we then created the GitHub repo, Terraform didn't also try to spin up another EC2 instance when we ran `terraform apply`. That is because while we had first_ec2.tf and github.tf in the same folder, all the resources in first_ec2.tf had already been created and were up and running, this information was stored in the state file. So Terraform checked this file and said "Oh the EC2 instance is already here, no need to create that. But the GitHub repo is not in the state file so I must still need to create that".

Once the GitHub repository was created its state was stored in the state file. So if we tried to run `terraform plan` it would say no changes need to be made as Terraform can see that all the requested infrastructure in the config files is up and running in the exact way the config files want it to be.

Blog About Contact   Zero to Devops

plan to recreate the EC2 instance. The code to create the AWS instance is still in the first_ec2.tf config file but its state is no longer in the state file so Terraform assumes it still needs to create it.

The state file is maintained by Terraform itself is not something you should ever really be directly editing.

___

Terraform Desired & Current State

Terraform's primary function is to create, modify, and destroy infrastructure resources to match the desired state described in a Terraform configuration.

The current state is the actual state of a resource that is currently deployed. Terraform will try to make sure that the deployed infrastructure is always based on the desired state defined by the .tf config files.

If there is a difference between the desired state and current state then `terraform plan` will present a description of the changes necessary to bring the current state in line with the desired state.

If the current state and desired state match then `terraform plan` will output that no changes are needed.

An important thing to note is that if something is not specified in the .tf config file then it is not part of the desired state.

For example in our AWS EC2 instance created at the start we have specified only the aspects that are mandatory to specify:

```
∨ resource "aws_instance" "my_ec2" {
    ami = "ami-05b622b5fa0269787"
    instance_type = "t2.micro"
  }
```

We have not for example specified what security group this should be part of. As such Terraform has put it in the default security group. However if I went on to the AWS Browser Console and manually changed the security group then the current state of the EC2 Instance would differ from what was in the state file. The state file will still show that the security group is default while the actual, physical current state of the EC2 security group is that it's a custom one (not default).

So you might think that if we now run `terraform plan` that Terraform will pick this change up and plan to change the security group back to the default. However that is not the case, even though the EC2 instance would differ from the state file it doesn't

Blog About Contact   Zero to Devops

As such all terraform will do when we run `terraform Plan` is request the current state of the EC2 see that it doesn't differ from the desired state and so not plan any changes, however it will see that it does differ from the state file so Terraform will update the state file to show the EC2 instance has a custom security group rather than the default.

Provider Versioning

During `terraform init`, if the version argument is not specified in the config file, the most recent provider plugin will be downloaded during initialisation.

For production use you should constrain the acceptable provider versions via configuration, this ensures that new versions with breaking changes will not be installed.

You specify this in the tf config file in the version field for the provider:

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 3.0"
```

There is syntax for describing versions:

`>=1.0` - Greater than or equal to the version

`<=1.0` - Less than or equal to the version

`~>2.0` - Any version in the 2.X range

`>=2.10, <=2.30` - Any version between 2.10 and 2.30

Dependency Lock File:
The Terraform dependency lock file (terraform.lock.hcl) allows us to lock to a specific version of the provider.
If a particular provider already has a version selection recorded in the lock file then Terraform will always re-select that version for installation, even if a newer version becomes available.

You can override that behaviour by adding `-upgrade` when you run `terraform init`

Attributes & Output Values
Note that the code written in this section is now in a separate folder to the EC2 and GitHub examples above.
Terraform can output the attributes of a resource that has been created.

Blog About Contact   Zero to Devops

~~Terraform.~~

For instance lets say you are creating an Elastic IP Address with Terraform. Once Terraform creates that EIP it can output the IP Address attribute and feed it into Terraform code that whitelists that IP within a security group.

In the below Terraform code we create both an EIP and an s3 bucket with outputs:
(Bare in mind this code is preceded by an AWS provider block)

```
resource "aws_eip" "lb" {
    vpc = true
}


output "eip" {
    value = aws_eip.lb.public_ip
}


resource "aws_s3_bucket" "mys3" {
    bucket = "zeal-tf-tutorial-hk112j"
}


output "mys3bucket" {
    value = aws_s3_bucket.mys3.bucket_domain_name
}
```

Here you can see we create an EIP with aws_eip and call it lb
Then we have the output block instructing Terraform to output the Public IP of the EIP we created. You can see that because we gave our EIP a terraform name of lb, we can reference that name with aws_eip.lb when asking for output of that particular resource. This comes in handy when you have multiple aws_eip resources in the script and want the output of one of them.

Next we create an s3 bucket with aws_s3_bucket and call it mys3, now we also specify a globally unique name for the s3 bucket within AWS.
Important to note that mys3 is the name we have given to this s3 bucket within our Terraform code but the buckets actual name on AWS will be zeal-tf-tutorial-hk112j.

Following on from the s3 resource block is the s3 output block. This is written to output the domain name of the bucket.

In the output blocks for the s3 bucket .bucket_domain_name is the **attribute** we want to output and .public_ip is the attribute to output for the EIP. If we don't specify a particular attribute at the end of the line then it would output all of the attributes for that resource.

Blog About Contact   Zero to Devops

mys3bucket (these are custom and can be whatever you want). One on each line followed by the actual output.

When we run `terraform apply` on this code we get the following output:

```
aws_eip.lb: Creating...
aws_s3_bucket.mys3: Creating...
aws_eip.lb: Creation complete after 1s [id=eipalloc-0c2af257bc22d5e11]
aws_s3_bucket.mys3: Still creating... [10s elapsed]
aws_s3_bucket.mys3: Creation complete after 10s [id=zeal-tf-tutorial-hk112k]

Apply complete! Resources: 2 added, 0 changed, 0 destroyed.

Outputs:

eip = "44.242.125.237"
mys3bucket = "zeal-tf-tutorial-hk112k.s3.amazonaws.com"
```

So the Terraform creates the EIP and s3 buckets as expected. Then it outputs the attributes we asked for.

You can find the attributes you can request to be outputted for a particular resource in the Terraform registry.

Referencing Cross-Account Attributes

Outputted attributes of resources can not only be used for user reference but also as inputs for other resources being created by Terraform.

In the below example we create an AWS Elastic IP and and an EC2 instance, we will use the outputted attributes of these two resources to allocate the EIP to the EC2.

```
resource "aws_instance" "my_ec2" {
    ami = "ami-05b622b5fa0269787"
    instance_type = "t2.micro"
}

resource "aws_eip" "lb" {
    vpc = true
}

resource "aws_eip_association" "eip_assoc" {
    instance_id = aws_instance.my_ec2.id
    allocation_id = aws_eip.lb.id
}
```

Blog About Contact   Zero to Devops

~~Second block we create the EIP and give it the name "lb".~~

The third block is what we want to look at here. We are creating a "resource" which is an aws_eip_assocation. This can be found in the AWS Documentation on the Terraform registry.
It will essentially allow us to automatically allocate the EIP to the EC2 we have created.

You can see the standard syntax of `resource "resource_type" "resource_name"` in the first line of the block.

In the next line instance_id we are clarifying the resource we want the EIP to attach to aws_instance.my_ec2 this refers to the instance we have created, however it requires the instance id, this is an attribute outputted once the EC2 is created. We can call this by adding the .id attribute reference at the end. So aws_instance.my_ec2.id
The final line allocation_id we are specifying exactly which EIP we want to allocate by its id. So aws_eip.lb.id, lb being the name we gave to the resource.

Run terraform apply on this and we now have an EIP and an EC2 running in AWS and the EIP is allocated to the EC2.

Terraform Variables Basics

Just like in any language repeated static values can create work in future if they need changing.
Lets say in you have a few Terraform projects that reference that same IP address 116.30.45.50 say this IP comes up 8 times across your Terraform projects.

That is fine when you are first writing it, but what happens a few months down the line and this IP needs changing? You have to go back in and change it 8 separate times which is effort spent and opens the door for human error.

Instead of repeating the IP value over and over again. When you first need it you can put the value in a central source from which you can import values in Terraform projects.
Now if you needed to update the value you can just update it in the central source and it will update in all your projects.

Blog About Contact   Zero to Devops

```
ingress {
    from_port = 80
    to_port = 80
    protocol = "tcp"
    cidr_blocks = ["116.30.45.50/32"]
}


ingress {
    from_port = 443
    to_port = 443
    protocol = "tcp"
    cidr_blocks = ["116.30.45.50/32"]
}
}
```

Here we are creating a simple AWS security group. The syntax to construct it isn't overly relevant, what is important is repetition of the IP address in the CIDR block. In production this might be repeated 10 or 20 times in various projects.

So what we do is create a separate .tf file to use as a central source for variables. In this case I have created a file called variables.tf and created a variable for this IP value.

```
1
2    variable "vpn_ip" {
3        default = "116.50.30.20/32"
4    }
```

No we can reference that variable directly in our cidr_blocks value with the syntax var.variable_name.

Blog   About   Contact   Zero to Devops

```
    ingress {
        from_port = 80
        to_port = 80
        protocol = "tcp"
        cidr_blocks = [var.vpn_ip]
    }


    ingress {
        from_port = 443
        to_port = 443
        protocol = "tcp"
        cidr_blocks = [var.vpn_ip]
    }
}
```

Terraform is smart enough to look for the vpn_ip variable and insert it where we have referenced it.

Now if we run `terraform plan` you can see that Terraform replaces the variable with the value we want (look at + cidr_blocks) :

```
# aws_security_group.var_demo will be created
+ resource "aws_security_group" "var_demo" {
    + arn                    = (known after apply)
    + description            = "Managed by Terraform"
    + egress                 = (known after apply)
    + id                     = (known after apply)
    + ingress                = [
        + {
            + cidr_blocks      = [
                + "116.50.30.20/32",
            ]
```

Now if you wanted to change the IP Address, instead of changing in multiple places in multiple projects you can just change it in the one variable file.

Various Ways to Assign Variables

There are multiple ways to assign values to variables:

- Environment Variables
-

Blog About Contact   Zero to Devops

~~Variable Defaults~~

Variable Defaults:

Variable Defaults are exactly as we have used in the IP example above, they are the default value you give to a variable when you create it.

Another example of this is the code below creating a very simple EC2 (as always this is preceded by a provider block):

```
resource "aws_instance" "my_ec2" {
    ami = "ami-05b622b5fa0269787"
    instance_type = var.instancetype
}
```

As you can see we have used a variable that we have called instancetype (though it could be called anything) and assigned it to the instance_type.

We have defined this variable as "t2.micro" in a separate file (t2.micro is a type of EC2 in AWS):

```
6    variable "instancetype" {
7        default = "t2.micro"
8    }
```

We have given the value "t2.micro" as the **default value** for the "instancetype" variable.

Unless explicitly told otherwise, when Terraform sees a reference to this variable it will supplement the default value we have specified above: "t2.micro"

A `terraform plan` on this code shows that Terraform does indeed swap var.instancetype for t2.micro:

```
# aws_instance.my_ec2 will be created
+ resource "aws_instance" "my_ec2" {
    + ami                          = "ami-05b622b5fa0269787"
    + 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"
```

This value t2.micro is the default value for the variable, we can override this in a couple of ways:

terraform plan -var= instancetype=t2.small

if we use this command for the code above here is the output:

```
# aws_instance.my_ec2 will be created
+ resource "aws_instance" "my_ec2" {
    + ami                          = "ami-05b622b5fa0269787"
    + 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.small"
```

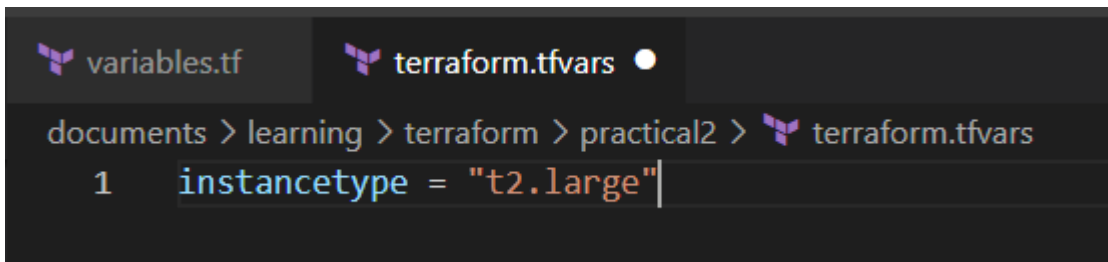You can see we have changed the value for the instancetype variable and now the EC2 will be created as type t2.small

If we ran `terraform plan` by itself again it would plan to create a t2.micro as that is still the default value of the variable.

This method is not best practice for overriding defaults

Overriding defaults with a terraform.tfvars file:
You can create a separate file called terraform.tfvars - it is always called exactly that.

Within this file you can specify different values from the default values you have already put in your variables file:

```
variables.tf        terraform.tfvars ●

documents > learning > terraform > practical2 >  terraform.tfvars
    1    instancetype = "t2.large"
```

Note that the syntax is different to defining the variable. This is closer to reassigning it, it still needs to have been initially defined though.

Now if we run `terraform plan`:

Blog About Contact   Zero to Devops

```
    + ami                          =  ami-05b622b5fa0269787
    + 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.large"
```

Terraform realises there is a terraform.tfvars file and looks there first for the instancetype variable value. If it weren't there it would use the default value "t2.micro" we defined in another file.

The terraform.tfvars file is not a replacement for the file where you define variables and give them default values, it is additional and allows you to specify values other than the default.

It is possible to pass a .tfvars file to use on the command line when you run `terraform apply.`

This is often used in production environments to have one config file that can be applied to multiple environments (UAT, PreProd, Prod). You would have your one config file and in order to provision or make changes to Prod you would pass the prod.tfvars file.

The command line syntax for this is as follows:
`terraform apply -var-file="prod.tfvars"`

Finally Environment Variables:
You can define a variable within your environment with the linux syntax:
`export TF_VAR_variablename="value"`

eg. `export TF_VAR_instancetype="t2.nano"`
Environment variables is the first place Terraform will look for values. So if instancetype has a value set in the environment that is the value Terraform will use even if it is also specified in the terraform.tfvars file.

Best Practice for variables in production is to define defaults in a variables.tf file and then put any necessary overrides in a terraform.tfvars file.

Data Types for Variables

Variable Types:

- **string**: a sequence of Unicode characters representing some text, like "hello".
- **number**: a numeric value. The number type can represent both whole numbers like 15 and fractional values like 6.283185.
- **bool**: a boolean value, either true or false. bool values can be used in conditional logic.
- **list** (or tuple): a sequence of values, like ["us-west-1a", "us-west-1c"]. Elements in a list or tuple are identified by consecutive whole numbers, starting with zero.
- **map** (or object): a group of values identified by named labels, like {name = "Mabel", age = 52}.

The type argument in a variable block allows you to restrict the type of value that can be assigned to the variable.

For example:

```
3    variable "my_id" {
4        type = string
5    }
```

Now the above variable my_id will only ever accept values with a type of string.

It is considered best practice to always specify the type that a variable will expect.

Using Maps and Lists with Variables

In the below code we are creating an EC2 and have two different variables we can use to choose the instance_type.

There is the "list" variable, a list of 3 instance types. There is also the "types" variable, a map of 3 more instance types.

Blog About Contact   Zero to Devops

```
18        instance_type = var.types
19    }
20
21    variable "list" {
22        type = list
23        default = ["m5.large", "m5.xlarge", "t2.medium"]
24    }
25
26    variable "types" {
27      type = map
28      default = {
29        us-east-1 = "t2.micro"
30        us-west-2 = "t2.nano"
31        ap-south-1 = "t2.small"
32      }
33    }
```

Currently you can see at the top instance_type is set to var.types but this corelates to the entire types map, 3 different instance types.
So when referring to a list or map how can we choose a specific value?

With **Maps** you can specifically choose a value by using its **key** with the following syntax:

```
16    resource "aws_instance" "my_ec2" {
17        ami = "ami-05b622b5fa0269787"
18        instance_type = var.types[us-east-1]
19    }
```

This will make the instance_type equal to the us-east-1 value in the types map. If you look at the code with the map itself, us-east-1 is the **key** for the "t2.micro" value.
As such instance_type in this case will be equal to t2.micro.

With **Lists** you can choose the value by index, exactly like Python.

```
16    resource "aws_instance" "my_ec2" {
17        ami = "ami-05b622b5fa0269787"
18        instance_type = var.list[0]
19    }
```

Index starts at 0 so in this case the instance_type will be set to the first item in our list "m5.large". Please note that the syntax we have used here var.list is because we have named the list of 3 instance types "list". This could easily be var.mylist or var.list1.

Blog About Contact   Zero to Devops

Terraform Count

The count parameter on resources can simplify configurations and let you scale resources by simply incrementing a number.

If you wanted to create two identical EC2 instances, one approach you could take is to use two separate resource blocks:

```
16    resource "aws_instance" "instance-1" {       16    resource "aws_instance" "instance-2" {
17        ami = "ami-05b622b5fa0269787"             17        ami = "ami-05b622b5fa0269787"
18        instance_type = "t2.micro"                18        instance_type = "t2.micro"
19    }                                             19    }
```

This approach can certainly be painful once you need lots of the same resource. What if we needed 5 EC2 instances? We don't want to copy and paste the block 5 times, especially if we will be changing anything in the instances, then we will have to make any changes to 5 different blocks.

What we can do is make use of the **count parameter**:

```
16    resource "aws_instance" "instance-1" {
17        ami = "ami-05b622b5fa0269787"
18        instance_type = "t2.micro"
19        count = 5
20    }
```

This simple addition tells Terraform to create 5 of this resource. If we now run `terraform plan` it will indeed plan to create 5 EC2 Instances of Type "t2.micro"

An important thing to note is that we have still only specified one name "instance-1" but terraform will create 5 resources. What it will do is append an indexing number to the given name for each resource it plans to create, so in this case the 5 EC2 instances in Terraform would be named: instance-1[0], instance-1[1], instance-1[2], instance-1[3] and instance-1[4].

The names above (instance-1[0] etc.) are just Terraform references so you can refer to a specific instance within Terraform. These names will not actually appear in the AWS console once they have been made. However it is important to note that some resources require actual names in AWS.

If a resource requires a unique value / identifier and you want to use the count parameter to build multiple of them, then you can use **count index** to make them

Blog About Contact   Zero to Devops

Below is an example using Terraform to create multiple IAM Users.

```
16    resource "aws_iam_user" "user" {
17        name = "user.${count.index}"
18        count = 5
19        path = "/system/"
20    }
```

The `${count.index}` syntax will append a number to the name of each IAM User
created by that block.

Here we can see the first two outputs of a `terraform plan` on the above code:

```
Terraform will perform the following actions:

  # aws_iam_user.user[0] will be created
  + resource "aws_iam_user" "user" {
      + arn           = (known after apply)
      + force_destroy = false
      + id            = (known after apply)
      + name          = "user.0"
      + path          = "/system/"
      + unique_id     = (known after apply)
    }

  # aws_iam_user.user[1] will be created
  + resource "aws_iam_user" "user" {
      + arn           = (known after apply)
      + force_destroy = false
      + id            = (known after apply)
      + name          = "user.1"
      + path          = "/system/"
      + unique_id     = (known after apply)
    }
```

 Notice how the name in each is unique: user.0 and user.1 the full output carries on for
all 5 IAM Users created.

You can probably already see that there will be some occasions where it would be far
more useful to have more specific names than user.1, user.2 etc. Well **Count Index** can
help with that as well. We can do this using a list variable, which I have put in the same
file for ease of screenshotting:

```
16    variable "iam_users" {
17      type = list
18      default = ["dev-user", "staging-user", "prod-user"]
19    }
20
21    resource "aws_iam_user" "user" {
22        name = var.iam_users[count.index]
23        count = 3
24        path = "/system/"
25    }
```

Blog About Contact   Zero to Devops

In the second block we are creating the IAM Users, we give them the terraform reference name of "user" but we give each one an actual name with:

```
name  =  var.iam_users[count.index]
```

Remember that specific items in lists can be referred to in Terraform via their index. So in the list iam_users above we have ["dev-user", "staging-user", "prod-user"] so var.iam_users[0] is equal to the first item in the list "dev-user".

With this in mind think that for each resource created **count.index** iterates once starting at zero. So the first IAM User created will have a name equivalent to var.iam_users[0] which as we have just seen equates to "dev-user". The next resource count.index will be equal to 1 so this IAM User will have a name equivalent to var.iam_users[1] or "staging-user".

Conditional Expressions

Conditional expressions select one of two Boolean values (true or false).

Structure of a conditional expression:

```
condition ? value_if_true : value_if_false
```

An example of when you might use this is having two resource blocks one for Dev and one for Prod, but you only need one depending on the environment you are building. You can use a variable called "istest" and when the variable is set to true Terraform should build from the Dev resource block and when it is set to false Terraform should build from the Prod resource block.

This is shown in the code below:

Blog About Contact   Zero to Devops

```
19        ami = "ami-05b622b5fa0269787"
20        instance_type = "t2.micro"
21        count = var.istest == true ? 1 : 0
22    }
23
24
25    resource "aws_instance" "prod" {
26        ami = "ami-05b622b5fa0269787"
27        instance_type = "t2.large"
28        count = var.istest == false ? 1 : 0
29    }
30
```

This looks kind of complicated but it really isn't.

First we have defined a variable called "istest", but currently it doesn't have a set value.

The first resource block is a standard block. Things are normal until the line beginning with count. Lets go over it:

```
count = var.istest == true ? 1 : 0
```

Remember the **count parameter** tells Terraform how many of the resource to make.

So line this essentially says if the istest variable is equal to true then count equals 1, else count equals 0. If istest is true make one of this resource, else make none.

The line has the exact same meaning as this python syntax:

```
1 ▾ if istest == True:
2        count = 1
3 ▾ else:
4        count = 0
```

Bare in mind though that unlike Python, Terraform does not capitalise Boolean expressions it's true & false not True & False

The second resource block has the same conditional expression just switched around. If the istest variable is set to false then this isn't a test environment and the prod environment should be built so count will be set to 1 in that resource block.

Now we will set the value of istest in a terraform.tfvars file to be true and run `terraform plan`:

Blog About Contact   Zero to Devops

```
+ 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
```

You can see that as "istest" is set to true Terraform is planning to create the Dev AWS instance and won't create the Prod one at all.

You can use higher numbers just like you normally would with count:

```
count = var.istest == true ? 4 : 0
```

If is test = true then make 4 of this resource.

Local Values

Local values assign a name to an expression, allowing it to be used multiple times within a module without repeating it.

In the code below we have defined the common_tags local value and have then used that in the following two resource blocks to minimise repetition with the syntax `tags = local.common_tags`

Note you define local values with the local**s** block but you call a local value with "local." (no S).

Blog About Contact   Zero to Devops

```
18              owner = "Devops Team"
19              service = "backend"
20         }
21    }
22
23
24    resource "aws_instance" "my_ec2" {
25         ami = "ami-05b622b5fa0269787"
26         instance_type = "t2.micro"
27         tags = local.common_tags
28    }
29
30    resource "aws_ebs_volume" "db_ebs" {
31         availability_zone = "us-west-2a"
32         size = 8
33         tags = local.common_tags
34    }
```

This is cool but it doesn't seem all that different to a standard variable. One of the main benefits of locals is that you can use expressions within them. Standard variables only take static values.

An example of using expressions to define a local value:

```
16    locals {
17         name_prefix = "${var.name != "" ? var.name : var.default}"
18    }
```

The expression here means If var.name **does not** equal an **empty string,** then name_prefix is equal to var.name, else name_prefix is equal to var.default.
Notice here how the expression is referring to two standard variables of var.name and var.default within the conditional expression. These two standard variables would be variables we have defined elsewhere.

This is really useful as without the ability to assign conditional expressions to a value we would have to hard code that expression into every block we need it in. With this we can just use the "name_prefix" local value instead.
Locals can also use Terraform Functions to define values (we haven't gone through them yet).

Blog About Contact   Zero to Devops

Terraform includes built-in functions that can be used to transform and combine values.

The general syntax for functions is a function name followed by comma separated arguments in parentheses:

```
function(argument1, argument2)
```

A specific example of this:

```
> max(5, 12, 9)
> Output: 12
```

The max function outputs the largest argument.

Terraform doesn't support user defined functions. Only those that are built-in to the language.

A complete list of supported functions can be found here: Terraform Registry - Functions

You can test out functions before using them in your Terraform files with the use of the `terraform console` command.

```
tclarke@DESKTOP-........~/documents/learning/terraform/practical2$ terraform console

Warning: Value for undeclared variable

The root module does not declare a variable named "istest" but a value was
found in file "terraform.tfvars". To use this value, add a "variable" block to
the configuration.

Using a variables file to set an undeclared variable is deprecated and will
become an error in a future release. If you wish to provide certain "global"
settings to all configurations in your organization, use TF_VAR_...
environment variables to set these instead.

> max(12, 20, 45)
45
> min(13, 16, 9)
9
> □
```

Type the command into your terminal and then you can test out functions to see how they operate and what their outputs will be.

Above we have tested both the max and the min functions.

Data Sources

Data sources allow data to be fetched or computed for use elsewhere in the Terraform configuration.

Blog About Contact   Zero to Devops

Every aws_instance resource block we have used so far has used a hardcoded ami.

```
17    resource "aws_instance" "my_ec2" {
18        ami = "ami-05b622b5fa0269787"
19        instance_type = "t2.micro"
20    }
21
```

Notice how the AMI (Amazon machine image) is hardcoded specifically to one AMI code.

This is the AMI for Ubuntu in the us-west-2 region. AMI codes change between regions even for the same type of AMI. So if we wanted to use the Ubuntu AMI in the us-east-1 region it would be a completely different code.

Remember that the region to deploy into is defined in the provider block:

```
10    provider "aws" {
11      region = "us-west-2"
```

So what will happen to our code if someone comes in and wants to change the region to us-east-1? Well they'd also need to change all of the references to region specific AMIs. They would also have to change any other region specific references for resources.

Data sources give us a work around for this:

```
17    data "aws_ami" "app_ami" {
18        most_recent = true
19        owners = ["amazon"]
20
21        filter {
22            name = "name"
23            values = ["amzn2-ami-hvm*"]
24        }
25    }
26
27    resource "aws_instance" "my_ec2" {
28        ami = data.aws_ami.app_ami.id
29        instance_type = "t2.micro"
30    }
```

You define the data source within the data block. So the data source is "aws_ami" and we have named the data source "app_ami".

The code within the data block is to specify which AMI type you want to call. For instance in this case we always want the Amazon Ubuntu AMI no matter the region (the AMI code will change between regions).

Blog About Contact   Zero to Devops

`owners` = [`Amazon`] - Specifies the 'owner' of the AMI. Basically the entity which created it, Amazon produce some of their own AMIs and we are using one of them.

`filter {` - Where you can filter the type down further eg. the exact instance type we want (Amazon Ubuntu).

`values = ["amzn2-ami-hvm*]` - specifying the exact instance type we want (you can find the AMI ID needed in the AWS docs)

We have then called this data source within the aws_instance resource block to get the correct AMI for whatever region we have specified in the provider block.

Now no matter what region we select Terraform will always pull the correct AMI for that region from the "aws_ami" data source.

Terraform Debugging

There are detailed logs in terraform which can be enabled by setting the TF_LOG environment variable to any value.
You can set TF_LOG to one of TRACE, DEBUG, INFO, WARN or ERROR to change the verbosity of the logs.

For instance in Linux you would set this variable on the Command Line to TRACE with:
`TF_LOG=TRACE`

This will print logs to the terminal and the different levels of verbosity will allow you choose the detail you would like to see the logs in.

You can output the logs to a file rather than output to terminal with the following command:
`export TF_LOG_PATH=/path/to-log-file`

TRACE is the most verbose, it is the default if TF_LOG is set to something other than a log level name.

To disable logging set the TF_LOG variable to empty:
`export TF_LOG=`

Terraform Formatting

You can use the `terraform fmt` command to rewrite your Terraform configuration files with best practice formatting.

Here is some example code for a simple EC2 instance in a file called my_ec2.tf before using `terraform fmt`

```
 1    terraform {
 2      required_providers {
 3        aws = {
 4          source = "hashicorp/aws"
 5          version = "3.33.0"
 6        }
 7      }
 8    }
 9
10    provider "aws" {
11      region = "us-west-2"
12      access_key = "ExampleKey"
13      secret_key = "SecretExampleKey"
14    }
15
16    resource "aws_instance" "my_ec2" {
17        ami = "ami-05b622b5fa0269787"
18        instance_type = "t2.micro"
19    }
20
```

Now here is the same code after inputting `terraform fmt my_ec2.tf` in the command line:

Blog About Contact   Zero to Devops

```
  3          aws = {
  4              source  = "hashicorp/aws"
  5              version = "3.33.0"
  6          }
  7        }
  8      }
  9
 10    provider "aws" {
 11      region     = "us-west-2"
 12      access_key = "ExampleKey"
 13      secret_key = "SecretExampleKey"
 14    }
 15
 16    resource "aws_instance" "my_ec2" {
 17      ami           = "ami-05b622b5fa0269787"
 18      instance_type = "t2.micro"
 19    }
 20
```

You can just use `terraform fmt` on it's own and not specify the file. This will format all config files in the directory.

Terraform Validate

`terraform validate` checks whether a configuration file is syntactically valid. It can check for issues including unsupported arguments, undeclared variables etc.

Here is some code with an invalid argument of "cheese":

```
 16    resource "aws_instance" "my_ec2" {
 17      ami           = "ami-05b622b5fa0269787"
 18      instance_type = "t2.micro"
 19      cheese = 12
 20    }
```

If we now run `terraform validate` on this file we get the following output:

Blog About Contact   Zero to Devops

```
 on new_ec2.tf line 19, in resource "aws_instance" "my_ec2":
   19:    cheese = 12

An argument named "cheese" is not expected here.
```

This allows you to quickly identify if there are any errors in your Terraform code.

Note that when you run `terraform plan` the command will also perform a validate to check that your code is valid before it makes the plan.

Load Order and File Structure

Terraform generally loads all config files within a directory in alphabetical order.
It will only load files ending with either .tf or .tf.json (this specifies the format in use).

When using Terraform in production generally you will have a few different config files for different aspects of the code.

Lets say we have the following code all in one config file called test.tf:

```
  4          source  = "hashicorp/aws"
  5          version = "3.33.0"
  6        }
  7      }
  8    }
  9
 10  ∨ provider "aws" {
 11      region      = "us-west-2"
 12      access_key = "ExampleKey"
 13      secret_key = "SecretExampleKey"
 14    }
 15
 16  ∨ variable "iam_user" {
 17      default = iam_user
 18    }
 19
 20  ∨ resource "aws_instance" "my_ec2" {
 21      ami             = "ami-05b622b5fa0269787"
 22      instance_type = "t2.micro"
 23    }
 24
 25  ∨ resource "aws_iam_user" "user1" {
 26      name = var.iam_user
 27      path = "/system/"
 28    }
```

In this test.tf file we have:
- Two provider blocks
- A variable block
- An EC2 resource block
- An IAM resource block

What we should do is split these blocks into their own relevant files:
- We will put the provider blocks into a file called providers.tf (all provider blocks would go here)
- We will put the variable block into a file called variables.tf (all variables would go here)
- We will put the EC2 resource block into a file called ec2.tf (all EC2 resource blocks could go in here)
-

Blog About Contact   Zero to Devops

We will remove the initial semantics.tf file

This leaves us with the below structure:
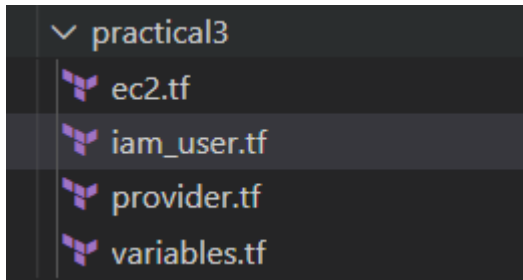
So if tomorrow there was a requirement to add another IAM User we could simply open up iam_user.tf and add the code in.

 This is good practice if you are not using modules. However if you are using modules within you terraform config (as you should in most production cases) there is a different practice.

Dynamic Blocks

These allow us to dynamically construct repeatable nested blocks, this is supported within resource, data, provider and provisioner blocks.

What we mean by nested blocks is that these are not top level, for instance in the example below the ingress blocks are nested within an AWS Security Group resource block.

Let's say we are defining a security group and we need to define multiple ingress rules:

```
 7   ingress {
 8       from_port = 9200
 9       to_port = 9200
10       protocol = "tcp"
11       cidr_blocks = ["0.0.0.0/0"]
12   }
13
14   ingress {
15       from_port = 8300
16       to_port = 8300
17       protocol = "tcp"
18       cidr_blocks = ["0.0.0.0/0"]
19   }
```

Here we only have two, but what if we needed 40 ingress rules to cover different ports? That would require 40 different blocks with the current syntax/structure of the code.

Blog About Contact   Zero to Devops

```
 2        type = list(number)
 3        description = "list of ingress ports"
 4        default = [8200, 8201, 8300, 9200, 9500]
 5    }
 6
 7    resource "aws_security_group" "dynamicsg" {
 8        name = "dynamic-sg"
 9        description = "Ingress for Vault"
10
11        dynamic "ingress" {
12            for_each = var.ingress_ports
13            content {
14                from_port = ingress.value
15                to_port = ingress.value
16                protocol = "tcp"
17                cidr_blocks = ["0.0.0.0/0"]
18            }
19        }
20    }
```

You can see in the first block we have defined the variable "ingress_ports" this is a list of ports.

The second block is a resource block for an AWS Security Group. Within that we have a nested dynamic block of type "ingress".

Within the dynamic block we have "for_each = var.ingress_ports"
This syntax is basically saying "for each item in the list ingress_ports" which is the list variable we have defined at the top.

So for each item in ingress_ports construct a new ingress inbound rule for the security group with the following values defined with in content {}

from_port = ingress.value
to_port = ingress.value

These are dynamic values, an ingress resource will be created for every item in the ingress_ports list. This ingress.value will represent each item in that list as it is iterated through.
So for the first iteration ingress.value will be 8200, next it will be 8201, then 8300 and so on for all the items in the list.

It might seem confusing that it is ingress.value and not ingress_ports.value. Doesn't the value refer to the name of the list variable we have created? So why isn't it referred to directly in the code?

Blog About Contact   Zero to Devops

iteration through ingress_ports to the Dynamic ingress block itself, so the ingress in
ingress.value refers to the actual dynamic block the code is in rather than the variable
the value is pulled from.

You can change this through the use of **iterator**.

```
 7    resource "aws_security_group" "dynamicsg" {
 8        name = "dynamic-sg"
 9        description = "Ingress for Vault"
10
11        dynamic "ingress" {
12            for_each = var.ingress_ports
13            iterator = port
14            content {
15                from_port = port.value
16                to_port = port.value
17                protocol = "tcp"
18                cidr_blocks = ["0.0.0.0/0"]
19            }
20        }
21    }
```

Notice how we define the iterator as port now instead of being ingress.value it is
port.value. This can make dynamic blocks easier to read and understand.

Finally the protocol and cidr _blocks will be the same for every ingress resource created
by the dynamic block so these can stay as static values.

So now if we ran `terraform plan` this one dynamic block will lead to 5 separate ingress
resources appearing in the plan. One for each port value in the ingress_ports list variable.

What if you want **multiple different dynamic values** in one dynamic block? For
instance say we wanted to alter the port value and cidr_blocks value. We couldn't use a
simple list, but we could use a list of maps:

```
1    variable "example" {
2        type = list()
3        description = "list in ports and cidrs"
4        default = [{"port"=8200, "cidr"="0.0.0.0/0"}, {"port"=8201, "cidr"="255.255.0.0/16"}]
5    }
```

This way we can define the values like:

`from_port = ingress.value["port"]`

`to_port = ingress.value["port"]`

`protocol = "tcp"`

`cidr_blocks = ingress.value["cidr"]`

Blog About Contact   Zero to Devops

Terraform Taint

Lets say we create a new EC2 resource in Terraform then users make a lot of manual changes both to the infrastructure and inside the server.
Now the EC2 resource will not match the Terraform code.

There are two ways to handle this:
- Import the changes to Terraform
- Delete and Recreate the resource

With the `terraform taint` command we can manually mark a Terraform-managed resource as tainted, this will force it to be destroyed and recreated on the next **apply**.

eg. `terraform taint aws_instance.my_ec2` note how this applies to a specific resource rather than a whole template.

The `terraform taint` command does modify the infrastructure, it merely alters its status in the state file to be 'tainted'. This marks it for destruction and recreation on the next `terraform apply`.

Why use taint?
When a resource declaration is modified, Terraform usually attempts to update the existing resource in place (although some changes can require destruction and re-creation, usually due to upstream API limitations).
In some cases, you might want a resource to be destroyed and re-created even when Terraform doesn't think it's necessary.

 Note that Terraform Taint has been deprecated in v0.15.2 onwards, now the best practice is considered to be using terraform apply with the "-replace" tag.
eg. `terraform apply -replace aws_instance.my_ec2`

Splat Expressions

Splat expressions (*) allow us to output attributes of multiple resources created by a single block.

The code below creates 3 IAM Users through the use of count and "count.index".

Blog About Contact   **Zero to Devops**

While the name in AWS will be iamuser.1 etc., the name we have given the resources in Terraform is lb. So to refer to each IAM resource created by this block individually within the Terraform code we would use lb[0], lb[1] & lb[2].

```
1   resource "aws_iam_user" "lb" {
2       name = "iamuser.${count.index}"
3       count = 3
4       path = "/system/"
5   }
6
7   output "arns" {
8       value = aws_iam_user.lb[*].arn
9   }
```

The second block of this code is the output block and this contains the **splat expression**. We have 3 IAM resources created by the first block all with the name "lb". In order to get the ARN (Amazon Resource Name) attribute for all three we could do:

value = aws_iam_user.lb[0].arn

value = aws_iam_user.lb[1].arn

value = aws_iam_user.lb[2].arn

Each line individually asks for the ARN of one of the IAM User resources we created. However this is quite repetitive, and if we needed 3 or 4 attributes the number of lines required would really add up.

So instead in the code we have used the splat expression * to essentially say give me ARN of all the IAM User resources with the name lb.

value = aws_iam_user.lb[*].arn

No when we apply the code we will receive an output to terminal with the ARNs of all three IAM resources we created in the first block.

Terraform Graph

The `terraform graph` command allows us to generate a visual representation of a configuration or execution plan.

The output of this command is in DOT format which can be converted into an image.

Blog About Contact   Zero to Devops

You can then use graphviz to convert that file into an image.

Ubuntu graphviz installation: `sudo apt install graphviz`
You can also download it directly from the graphviz website on windows.

Example output of `terraform graph` in image format:

Saving Terraform Plan to File

You can save the output of `terraform plan` to a file.
Then you can specify this file when using `terraform apply` to be certain that only the changes shown in the plan file will be made and no others.

Syntax:
`terraform plan -out=`*`filepath`*

Then to use the file with apply, be in the directory with your plan file:
`terraform apply filename`
Where "filename" is replaced with whatever you named your plan file.

This feature is very useful and often used in production when working with Terraform in conjuction with CI/CD.

Terraform Output

The `terraform output` command is used to extract the value of an output variable from the state file.

Lets use the splat expression example again:

Blog About Contact   Zero to Devops

```
 3        count = 3
 4        path = "/system/"
 5     }
 6
 7  output "arns" {
 8        value = aws_iam_user.lb[*].arn
 9  }
```

When you run `terraform apply` on this it will create the resources but due to the output block it will also output to terminal:

```
iam_arn = [
    "arn:aws:iam::795574780076:user/system/iamuser.0"
    "arn:aws:iam::795574780076:user/system/iamuser.1"
    "arn:aws:iam::795574780076:user/system/iamuser.2"
]
```

Now what if you wanted to see this output again a few days later? Well for that you can use `terraform output`:

```
terraform output iam_arn
```

This will print to terminal the exact same output as above.

Alternatives:

Alternatively to see the output again you could run another `terraform apply` as even if there are no changes to make this command will still provide the outputs specified in the code. Or you could directly look at the state file.

Terraform Settings

There is a special terraform configuration block type which is used to configure behaviours of Terraform itself.

Terraform settings are gathered into terraform blocks:

```
11  terraform {
12      required_version ="> 0.12.0"
13  }
```

The required_version setting specifies which versions of Terraform can be used with your configuration. This means if the Terraform version used to run this code is not

Blog About Contact   Zero to Devops

Another useful setting is required_providers, you will have seen this one in this guide multiple times already:

```
1  v terraform {
2  v    required_providers {
3  v       aws = {
4            source  = "hashicorp/aws"
5            version = "3.33.0"
6          }
7       }
8     }
9
```

This specifies all of the providers required by the current module and maps each local provider name to a source address and version constraint.

Handling Large Infrastructure

When you have a large infrastructure you will inevitably face issues related to API limits for a provider.

To help solve this you can switch to smaller configurations which can each be applied independently.
So instead of having one file containing all your VMs, DBs and Security Groups, Networking infrastructure etc.

You can actually split this out into separate .tf files where each can be independently applied.

Example of smaller config benefits:
For example when you use `terraform plan` the command has to refresh the state of each physical resource in the config file from your provider's API to see if any changes need to be made.

Now say you have all of your infrastructure in one big .tf file. You make some changes to the EC2 instances and nothing else, well now when you use `terraform plan` Terraform

Blog About Contact   Zero to Devops

If instead you had split out all the resources into appropriate files and the needed to make EC2 changes. Then you can go into ec2.tf (or whatever you named it) and make those changes, now when you can run `terraform plan ec2.tf` and Terraform will only have to request the state of the resources in ec2.tf, which is **far less API calls**.

Specify the target resource type to update:
If you do have a really large config file then you can use the target flag to specify the resources you want refreshed:
Syntax: `-target=resource`

For example:
`terraform plan -target=ec2`
This will allow you to operate on an isolated portion of a very large config file. With this flag the `terraform plan` command will only refresh the state of the EC2 instances.

Prevent refresh altogether:
You can use the refresh flag to prevent `terraform plan` refreshing the state of resources when making a plan. This will do an "update in place"

syntax:
`terraform plan -refresh=false`
You can use this when you want a plan but don't need to refresh the state of the resources.

Important for the exam to note is that when you do this the resources in the plan will be preceded by a yellow tilde mark `~`
This tilde mark lets you know that the resources have been updated in place and not refreshed.

Zipmap Function
The zipmap function constructs a map from a list of keys and a corresponding list of values.

List of Keys	List of Values	Zipmap

| orange | orange | orange=orange |
| strawberry | red | strawberry=red |

An example of this in action:

```
tclarke@DESKTO          :~$ terraform console
> zipmap(["pineapple", "oranges", "strawberry"], ["yellow", "orange", "red"])
{
  "oranges" = "orange"
  "pineapple" = "yellow"
  "strawberry" = "red"
}
```

Here you can see we are making use of the `terraform console` command which allows us to test Terraform functions in the terminal.

We have provided zipmap with two lists as in the table above. It has then "zipped" those two lists into a map of key-value pairs.

Example of real usage:

1. We have the below code which constructs 3 IAM Users and then outputs the ARN and Name for each of them:

```
16    resource "aws_iam_user" "lb" {
17        name = "iamuser.${count.index}"
18        count = 3
19        path ="/system/"
20    }
21
22    output "arns" {
23        value = aws_iam_user.lb[*].arn
24    }
25
26    output "name" {
27        value = aws_iam_user.lb[*].name
28    }
```

2. When we run `terraform apply` we get the following output:

```
Outputs:

arns = [
   "arn:aws:iam::532952094995:user/system/iamuser.0",
   "arn:aws:iam::532952094995:user/system/iamuser.1",
   "arn:aws:iam::532952094995:user/system/iamuser.2",
]
name = [
   "iamuser.0",
   "iamuser.1",
   "iamuser.2",
]
```

As you can see this gives us two lists, one list of the IAM Users ARNs and a second list of IAM Users Names

3. What we can do instead is combine these outputs into a Map with zipmap:

```
16    resource "aws_iam_user" "lb" {
17        name = "iamuser.${count.index}"
18        count = 3
19        path ="/system/"
20    }
21
22    output "combined" {
23        value = zipmap(aws_iam_user.lb[*].arn, aws_iam_user.lb[*].name)
24    }
```

Notice how we have replace the two original output blocks with just one. Where we have used zipmap on the original two values.

4. The output of this is:

```
Apply complete! Resources: 0 added, 0 changed, 0 destroyed.

Outputs:

combined = {
   "arn:aws:iam::532952094995:user/system/iamuser.0" = "iamuser.0"
   "arn:aws:iam::532952094995:user/system/iamuser.1" = "iamuser.1"
   "arn:aws:iam::532952094995:user/system/iamuser.2" = "iamuser.2"
}
```

Now we have both the ARNs and Names of each resource mapped to each other in one simple output.

Blog About Contact   Zero to Devops

Up until now all examples and information have centred around creation and destruction of infrastructure scenarios.

Lets say we create an EC2 instance with Terraform, it is just an instance and nothing else. What if we want to have software installed on it automatically?

Provisioners are used to execute scripts on a local or remote machine as part of resource creation or destruction.

For example:

We can code in our config file that when we create a new EC2 instance, Terraform should also execute a script which installs Nginx web-server.

Types of Provisioners:

There are two main types of provisioners: **local-exec** and **remote-exec**.

Local Exec:

These allow us to invoke local executable after a resource is created, what this means is that a local-exec provisioner can run an executable (command) on the machine that is applying the Terraform (local):

```
16    resource "aws_instance" "my_ec2" {
17      ami          = "ami-05b622b5fa0269787"
18      instance_type = "t2.micro"
19
20      provisioner "local-exec" {
21        command = "{aws_instance.web.private_ip} >> private_ips.txt"
22      }
23    }
```

In this code we are creating an EC2 instance and within the EC2 resource block we have a Provisioner block. This is a local-exec provisioner that will run a command that places the Private IP of the instance created by the resource block in a file on the local machine called private_ips.txt

This is what a local-exec does, it allows Terraform to run commands on **the machine that is applying** the Terraform.

Local provisioners are really powerful and one of their most important uses is to run ansible playbooks once resources have been created.

Blog About Contact   Zero to Devops

on a **remote** server.

Implementing Remote-Exec Provisioners

Below is an example of the code you would use to have a remote-exec install Nginx on an EC2:

This code has 3 blocks: A resource block, a nested provisioner block and a nested connection block. Nested meaning that they are within the resource block.

```
16    resource "aws_instance" "my_ec2" {
17      ami            = "ami-05b622b5fa0269787"
18      instance_type = "t2.micro"
19      key_name = "TF-keys"
20
21      provisioner "remote-exec" {
22        inline = [
23          "sudo amazon-linux-extras install -y nginx1.12",
24          "sudo systemctl start nginx"
25        ]
26      }
27
28      connection {
29        type = "ssh"
30        user = "ec2-user"
31        private_key = file("./TF-keys.pem")
32        host = self.public_ip
33      }
34    }
```

1. Resource Block - This is pretty standard from what we've seen so far. It does however have the addition of key_name. This is because remote exec requires access to the EC2 being created, it can either do this with a user, password or via an access key pair over SSH. We are using the latter and so the resource block needs to know the key name we are using. This is just the name of the key pair within AWS:

Blog About Contact   Zero to Devops

2. Provisioner Block (nested) - We have specified this as a "remote-exec" provisioner block, this is the block that will run our remote commands. Then we are using "inline []" this allows us to specify a list of commands to be run by the provisioner on the EC2 we are creating. These commands should be listed in order and you can see we have 2 commands: 1 to install Nginx and 2 to start Nginx.

3. Connection Block (nested) - This is the block that specifies how the provisioner block should connect to the EC2 instance. Within this we have 4 elements defined:

   - `type = "ssh"` - We want to use SSH to connect
   - `user = "ec2-user"` - Every Amazon Machine Image has a default user. With the Linux AMI we are using the default is "ec2-user", it defines what user the remote-exec should run commands from.
   - `private_key = file("./TF-keys.pem")` - We use this to tell the connection block what the access key is to the EC2. When you create a key pair within AWS it automatically gives you .pem file with your keys. We have put a copy of this file in the same directory with our TF config files. Now we can use the file command with the filepath of "./TF-keys.pem". This access key is used for all EC2s on the AWS account.
   - `host = self.public_ip` - This tells the provisioner the address of the instance we want to connect to. Connection blocks can use the method self to refer to the parent resource block. So this basically says the host address is the public IP of the parent resource of the connection block.

If you run `terraform apply` with the above code it will create an AWS EC2 Instance and then proceed to install Nginx and start Nginx up. If it loops when "connecting to remote host via SSH" and never connects this is likely because you haven't configured SSH on port 22 to be open in your EC2s security group, you can do this manually in the AWS console if just trying to get this example to work.

Creation Time & Destroy Time Provisioners

Creation-time provisioners are only run during creations, not during updating or any other lifecycle. If a creation-time provisioner fails, the resource is marked as tainted. This

Blog About Contact   Zero to Devops

Destroy-time provisioners are run before the resource is destroyed.

The code below shows an example:

```
16  ∨ resource "aws_instance" "my_ec2" {
17      ami             = "ami-05b622b5fa0269787"
18      instance_type = "t2.micro"
19      key_name = "TF-keys"
20
21  ∨   provisioner "remote-exec" {
22  ∨     inline = [
23          "sudo apt install nano"
24        ]
25      }
26
27  ∨   provisioner "remote-exec" {
28        when = destroy
29  ∨     inline = [
30          "sudo apt remove nano"
31        ]
32      }
```

Here we are creating an EC2 resource with two remote-exec provisioner blocks. One is a standard creation-time and the other is destroy-time. Upon running `terraform apply` this EC2 resource will be created and then the provisioner block will install Nano on the EC2 instance.

The when within the destroy provisioner allows us to specify that it is a destroy-time provisioner. If this is not included it is assume the provisioner is creation-time.

When we then run `terraform destroy` it will first remove Nano before destroying the instance.

Note that below these two provisioner blocks is also a connection block that isn't shown.

On failure:

You can use on failure in creation time provisioners. It essentially gives the code instructions on what to do if the command fails:

```
 23        sudo apt install nano
 24      ]
 25      | on_failure = continue |
 26    }
```

on_failure can either be set to continue or fail. If it is set to continue then even if this code failed to install Nano the resource will not be marked as tainted and will carry on as normal. If it is set to fail then the normal behaviour of marking the resource as tainted will occur, fail is the default if not specified.

The DRY Principle & Modules

In software engineering don't repeat yourself (DRY) is a principle of software development aimed at reducing repetition of software patterns.

Terraform helps us be DRY by using modules. Modules allow us to centralise the structure of resources so we can call them into our config files easily. For instance you will have seen in this guide that we have used the same EC2 resource block many times in different files.

Here is an example EC2 resource block with tags and a security group specified:

```
16    resource "aws_instance" "my_ec2" {
17      ami            = "ami-05b622b5fa0269787"
18      instance_type = "t2.micro"
19      security_groups = ["${aws_security_group.mysg.name}"]
20
21      tags {
22          Name = "web-server"
23      }
24    }
```

Rather than including the above in every single config file we create we can instead put this code in a central location and call it in a module to be used in any tf file we want. Lets say we put the above code in a directory called ec2 and that in a directory called modules (the names aren't relevant). Then we can call it in another file like so:

```
 27    module "my_server" {
 28        source  = "../../modules/ec2"
 29    }
```

Blog About Contact   Zero to Devops

can then be used in multiple Terraform projects. In this way we don't have to write out the same resource blocks over and over again in each project.

This is different to count as count allows us to avoid repeating ourselves within a config file and modules allow us to avoid repeating ourselves between two or more different projects (different sets of infrastructure or environments).

When you use modules in a project you will need to use `terraform init` to install those modules in the project. This is because the modules you create you will store in a central source which is a separate directory or remote repository to the project you'll be working on.

Variables and Terraform Modules

It is common in infrastructure management to build multiple environments, eg. one for staging, production, testing etc. With these you will want a similar setup but will want to change some variables.

For example in your staging environment you may want a small EC2 instance, but in your production you may want a large EC2 instance.

If you have hardcoded details in your module you will not be able to change it within a project you use it in:

```
16    resource "aws_instance" "my_ec2" {
17      ami             = "ami-05b622b5fa0269787"
18      instance_type = "t2.micro"
19      security_groups = ["${aws_security_group.mysg.name}"]
20
21      tags {
22        Name = "web-server"
23      }
24    }
```

Lets say we have 'Project A' and in Project A we use the above module. This EC2 module is a t2.micro, within Project A there is nothing we can do to change that. We can not specify a different instance size as it is hard coded into the module.

However we can use variables within modules which we can then alter from within Project A:

Blog About Contact   Zero to Devops

```
19      security_groups = ["${aws_security_group.mysg.name}"]
20
21      tags {
22          Name = "web-server"
23      }
24  }
```

Here you can see we have now changed the EC2 module to have instance_type set to a
variable called type_of_instance.

We have also created a separate file within the same directory as the EC2 module called
variables.tf. Within that we have given a default value to the type_of_instance variable in
case the user does not specify:

```
26      variable "instance_type" {
27          default = "t2.micro"
28      }
```

Now within Project A when the user calls this EC2 Module they can specify the variable
value within the module block:

```
32      module "my_server" {
33          source        = "../../modules/ec2"
34          type_of_instance = "t2.large"
35      }
```

Now if we ran `terraform plan` we would get a plan output for an EC2 instance of type
t2.large but where all other details specified in the EC2 module are the same.

Terraform Registry & Modules

The Terraform Registry has a library of pre written modules that you can browse. So
rather than writing one from scratch you can look for a module on the registry that is a
close fit and alter it to your liking. The majority of these are community modules written
by the community.

The registry also has verified modules which are maintained by third party providers like
AWS, Azure etc.

Blog About Contact   Zero to Devops

Terraform Workspace

Terraform allows us to have multiple workspaces, within each workspace we can have a different set of variable values. You can have the same infrastructure in two different workspaces and each workspace would be isolated with its own variable values.

For instance you could have the variable type_of_instance.

Then if you have two environments put in two workspaces:

Workspace A - Staging Environment: type_of_instance = t2.micro
Workspace B - Production Environment: type_of_instance = t2.large

Workspaces allow you to have the same variable set to different values dependent on the workspace you are in. In the above example the value of type_of_instance changes dependent on the workspace that it is in. So if we were in Workspace A , then when we ran `terraform plan` it would plan to create an EC2 of type t2.micro. Whereas in Workspace B it would create one of type t2.large.

Terraform Collaboration | Remote Repositories

Most enterprise usage of terraform will involve collaboration through a central GIT repository like GitHub or GitLab.

When sharing Terraform code it is never a good idea to push your .tfstate files to a remote repository as they often contain sensitive information.

Source Argument:

You can use Modules collaboratively through remote repositories using the source argument of the module block:

```
32   module "my_server" {
33     source  = "git::https://github.com/username/mymodule.git"
34     type_of_instance = "t2.large"
35   }
```

The source argument can either be for a remote or local module. In the above code the source we are using is for a remote GitHub repository. Note that when using a git repository URL whether via SSH or HTTPS the full format is for the URL to be preceded by `git::`.   However Terraform is smart enough to understand it is a git repo even if this is left out.

Ref Argument to pull specific branch:

```
32   module   my_server   {
33     source  = "git::https://github.com/username/mymodule.git?ref=branch1"
34     type_of_instance = "t2.large"
35   }
```

Terraform & .gitignore:

You may want to ignore the following files when adding your Terraform code to Git:

- Terraform directory itself as this is created when `terraform init` is run.
- terraform.tfvars which is likely to contain sensitive informaiton. However this is dependent on how you have set it up.
- terraform.tfstate secrets (passwords, keys) are stored in plain text in the tfstate file so you don't want to commit them.
- crash.log

Terraform Remote Backend

Terraform has a feature called remote backend which can be used to store a tf state file if you need to do remote state management. This will allow you to refer to a remote TF State file within your terraform code. For example you can store your state file in an S3 Bucket.

This is useful as it means that everyone in a team can work with the same state file. This prevents unwanted issues like two people working with slightly different state files or trying to perform a Terraform Apply at the exact same time and breaking the infrastructure.

With one shared state file every time an apply is done the state file is locked so two apply processes can not be run at the same time.
Using a remote backend for state files is the recommended process for collaborative environments. Nearly all businesses using Terraform use remote backend for state files.

Below is an example of the code you might use to establish an s3 remote backend for your Terraform state files.

Blog About Contact   Zero to Devops

```
4            key = "path/to/mykey"
5            region = "eu-west-1"
6        }
7    }
```

State Locking

Whenever you are performing a write operation Terraform locks the state file. This is extremely useful when collaborating on Terraform in a team.
This is important as during an ongoing `terraform apply` if others also do an apply it can corrupt your state file.

- For example Person A is terminating an RDS resource which has an associated rds.tfstate file
- Person B is also at the same time trying to resize the RDS resource.

This will cause issues in the tfstate file which may lead to corruptions. For this exact reason during a `terraform apply` the state files are locked so two contradictory actions cannot be run at the same time.

This is also why it is important to have a shared remote backend for state files when working as a team. This way everyone works off the same state files rather than local copies, so if an apply is occurring the one state file the entire team uses is locked until the apply is done.

When using a remote backend you will need to create a lock for the Terraform to use. This can be done in AWS with DynamoDB, if this lock is not created then there can still be issues with simultaneous applies and corruptions

Terraform State Management

There may be some cases where we need to modify the Terraform state. It is important to never modify the state file directly, instead we can make use of the `terraform state` command.

Sub Command	Description
list	List resources within the Terraform state file
mv	Moves item with terraform state (can use for renaming)
pull	Manually download and output the state from remote state
push	Manually upload a local state file to remote state
rm	Remove items from the terraform state
show	Show the attributes of a single resource in the state

Using Terraform Import

There may be a time when there is a resource which has already been created manually and not in Terraform. If we wanted to change something in that resource we would have to do it manually.

Ideally we can bring that existing resource into infrastructure as code.

To do this we can use `terraform import`

However this command will only create a state file for the manual resource it will not create the .tf file so we can edit the code in future.

To use terraform import correctly we have to write the Terraform code for the manual resource in a tf file ourselves and then link that code to the manual resource when we use terraform import.

Terraform Cloud

Terraform Cloud manages "Terraform runs" in a consistent and reliable environment with various features like access controls, private registry for sharing modules, policy controls and others.

Blog About Contact

Zero to Devops

Blog About Contact

Zero to Devops

Blog About Contact   Zero to Devops

AWS Solutions
Architect Associate -
Everything You Need   Bash Scripting Cheat
To Know   Sheet

Blog About Contact

Zero to Devops

in ✉

in ✉

Blog About Contact

Zero to Devops

in ✉