KP-Labs Terraform Udemy Course Notes

Prerequisites:

Terraform CLI tool installed - https://learn.hashicorp.com/tutorials/terraform/install-cli

AWS Account - https://aws.amazon.com/premiumsupport/knowledge-center/create-and-activate-aws-account/

AWS CLI - https://aws.amazon.com/cli/

Note:

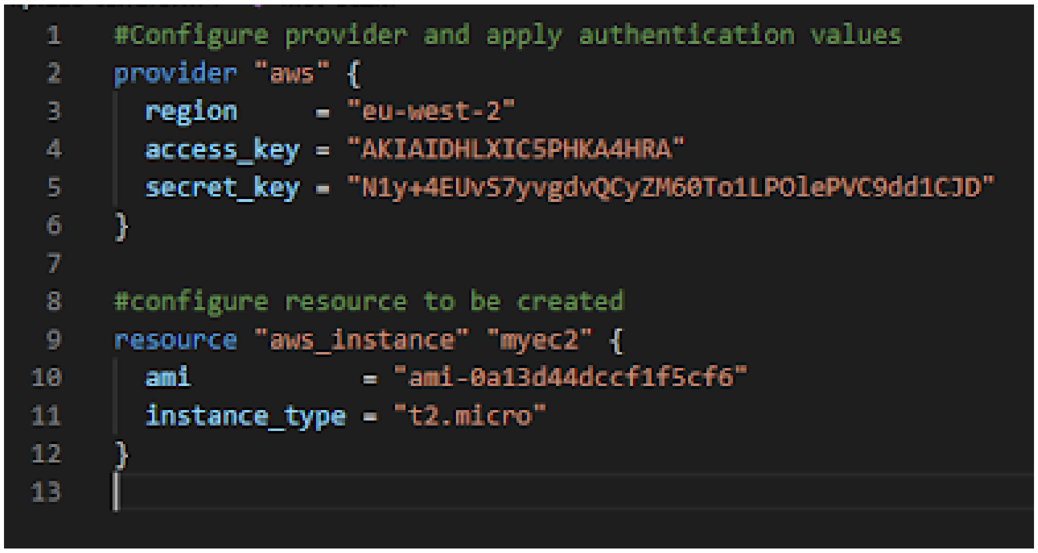
- All code shown in these notes will be used for deploying infrastructure to AWS under the free tier
- The AWS CLI will not be needed until later, where it's shown you can configure it as part of security best practices
- Though AWS is the preferred provider for these notes, the same concepts and skills are still applicable to other cloud providers including Azure and GCP; see the respective cloud provider's documentation on https://registry.terraform.io/browse/providers

3.0 - Deploying Infrastructure with Terraform

3.1 - EC2 Instance Creation

- Terraform supports a significant number of providers, this becomes an important consideration for organisations when working with terraform.
- Before using a provider, make sure to be logged into it
- In the case of AWS, can make use of any of the following:
    - Static credentials
    - Environment variables

- Shared credentials files
- EC2 Roles

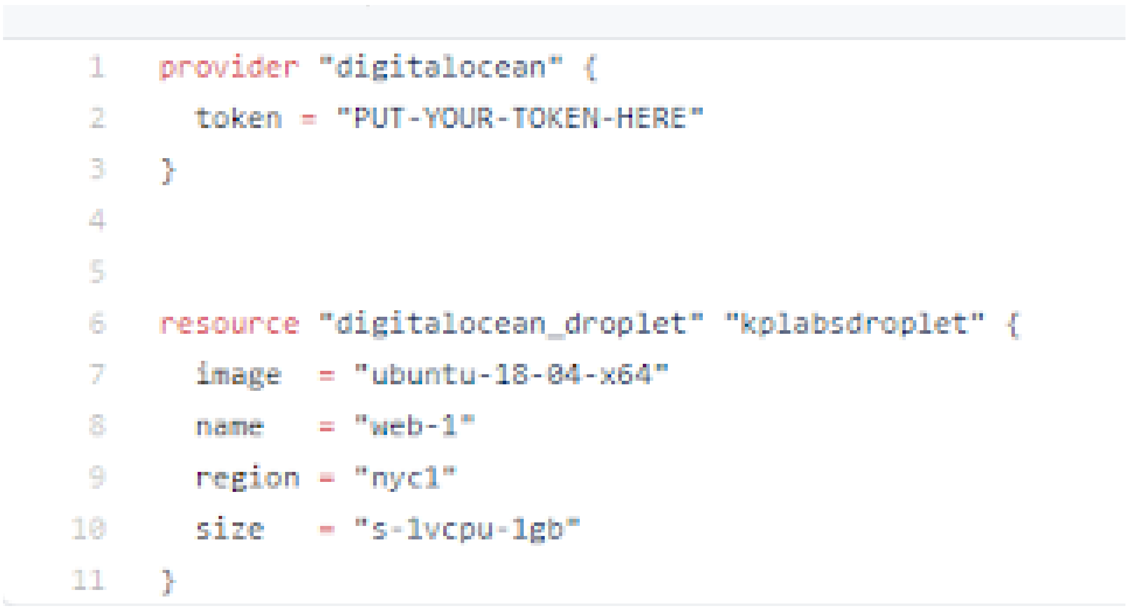
- For this demo, using static credentials:
  - AWS Console -> IAM -> Create User -> Allow programmatic access -> create user -> access and secret keys shown
- Add following to first_ec2.tf file:
  - 

```
#Configure provider and apply authentication values
provider "aws" {
  region     = "eu-west-2"
  access_key = "AKIAIDHLXIC5PHKA4HRA"
  secret_key = "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"
}

#configure resource to be created
resource "aws_instance" "myec2" {
  ami           = "ami-0a13d44dccf1f5cf6"
  instance_type = "t2.micro"
}

```

  - The above code first configures the provider, aws, providing the region location for the vm deployment, along with the access and secret keys
  - The code then looks to create an aws resource, in this case an ec2 instance of a particular ami and instance type
    - Instance type specifies parameters such as memory and cpu
  - Note: ami and instance_type are the only two required parameters for a aws_instance resource, there are optionals available
  - To initialize: **terraform init**
    - Terraform initialises the current directory, downloading additional plugins required; which vary for each provider
  - To validate the syntax used in config files: **terraform validate**
  - To format: **terraform fmt**
  - To plan the execution of the configuration: **terraform plan**
  - To apply: **terraform apply**
  - To destroy: **terraform destroy**

3.2 - Providers and Resources

- Terraform is capable of supporting multiple providers
- The provider details the infrastructure is to be launched on must be specified in the configuration files
- In some/most cases, authentication tokens will also be required
- When the command terraform init is ran, terraform downloads plugins associated with the provider
- Each provider has different resources that can be created, with each resource type specific to the particular provider e.g. for AWS:
    - Aws_instance - Virtual Machine/Instance
    - Aws_alp - Application load balancer
    - Iam_user - Identification Application Managment User
- The above resources couldn't be provisioned by e.g. Azure with this syntax, would have to use the provided syntax for Azure
- Consider the following example:



```
provider "digitalocean" {
  token = "PUT-YOUR-TOKEN-HERE"
}


resource "digitalocean_droplet" "kplabsdroplet" {
  image  = "ubuntu-18-04-x64"
  name   = "web-1"
  region = "nyc1"
  size   = "s-1vcpu-1gb"
}
```

- In terms of syntax, the general format is followed, defining a provider and configuring it, followed by a resource which is configured as desired.

    - Since this is for a different provider (DigitalOcean), the syntax for parameters such as the image or region names
    - In terms of authentication, only one token is required for provider configuration unlike AWS.

3.3 - Destroying Infrastructure

- Obviously we don't want to keep infrastructure running forever and racking up charges
- To bring down infrastructure, can run the command terraform destroy
- If you wish to destroy a specific target, add the -target flag in the format:
    - Terraform destroy -target resource_Type.resource_name
    - E.g. terraform destroy -target aws_instance.ec2
- Alternatively could comment out the resource you don't wish to be applied or destroyed, not recommended in general practice
- Note: to automatically destroy: terraform destroy --auto-approve

3.4 - Terraform State Files

- Terraform keeps track of the infrastructure being created in a state file
- Allows terraform to map real-world resources to existing configurations
- Includes resource details including instance ID, Ip addresses etc.
- Therefore, when terraform destroy is applied to a particular target, all the details relating to that resource will be removed from the state file
- State file always named terraform.tfstate
- If changes made outside the terraform configuration files, when terraform apply is ran again, the tfstate file will check to make sure that the real world configuration is set up correctly, if not it will apply the appropriate changes to return/update to the desired state

3.5 - Desired and Current States

- Desired state:
  - The state defined within a terraform configuration's resource block
  - E.g. desired state for the EC2 state: instance_type = "t2.micro"
- Current state:
  - The state defined within the terraform.tfstate file
  - The state of the resource which is currently running
- To update the current state, run ***terraform refresh***
  - The current state is always refreshed during a terraform plan or application
  - Terraform will always apply changes to ensure that the current state matches that of the desired state

3.6 - Challenges of the Current State & Computed Values

- If not running terraform in a system with a readily-available editor, can run the command terraform show to view the information contained within the current state file
- Note: suppose a change is made outside the desired state that wasn't specified e.g. add a security group, terraform will not act to update the current state if it's not already specified in the desired state

3.7 - Terraform Provider Versioning

- Terraform providers exist to provide a link between terraform and the service provider, allowing terraform to provision infrastructure to the appropriate provider
- Plugins for the providers are released separately from terraform and are updated in their own time
- In terraform init, the latest provider plugin will automatically be downloaded if the version isn't specified
    - In production, it's recommended to specify the version that you know the code works for e.g. aws version 2.7
- To specify the provider version, in the provider block of the configuration file, add version = "version number" as an exact number or express it as a criteria:

Version Argument	Description
>=x.y	Greater than or equal to version value
<=x.y	Less than or equal to version value
~>x.y	Any version in the subrange of value x i.e. any version of 2.y
>=a.b,<=c.d	Any version between a.b and c.d

- Note, certain provider plugins aren't compatible with particular versions of terraform, in this case terraform will specify alternative versions to try

3.8 - Types of Providers

Terraform providers are generally split into 2 categories:

- Hashicorp distributed - Automatically downloaded during terraform init
- 3rd Party

The need for third party providers arises whenever an official provider doesn't support a particular functionality, or when organizations have developed their own platform to run terraform on.

Hashicorp distributed providers are listed under "Major Cloud Providers" under the HashiCorp website; third party providers are under the "Community" tab.

When attempting to initialise with a third party provider, it's likely that an error will occur. As mentioned, terraform init cannot automatically install the plugins for third party providers, they must be installed manually. The installation can be achieved by placing the plugins into the system's user plugins directory.

Operating System	User Plugins Directory
Windows	%APPDATA%\terraform.d\plugins
Other Systems	~/.terraform.d/plugins

4.0 - Read, Generate and Modify Configurations

4.1 - Attributes and Output Values

Terraform is capable of outputting the attribute of a resource with the corresponding values.

The outputted attributes can be used for both user reference as well as an input to other resources being created. For example once an Elastic IP address (EIP) is created, it should automatically be added to the security group to be whitelisted.

For an output, if you set a value as the resource_Type.resource_id, this will output all the attributes associated with that resource. For a particular piece of information, the value should be: resource_type.resource_id.attribute_name; such as aws_eip.lb.public_ip.

Note: A list of attributes which can be exported/output is listed on the HashiCorp website

In general:

- Attribute - Value associated with a particular property of a resource
- Outputs - Used to output the value of a particular attribute

4.2 - Referencing Cross-Account Resource Attributes

As suggested previously, when creating resources you should be able to use attributes and outputs to allow automatic configuration, two examples will be considered in this section:

1. Creating an EIP and assigning it to an AWS EC2 instance
2. Creating an EIP and assigning it to a security group for whitelisting

4.2.1 - Example 1: EIP Association to EC2 Instance

To associate the EIP with the EC2 instance, need to add a new resource called “aws_eip_association”. This resource must specify the instance ID and the IP address allocation’s ID.

Values to be specified in the form:

- `instance_id = aws_instance.instance_id.id`
- `allocation_id = aws_eip.eip_id.id`

4.2.2 - Example 2:

When specifying the security group as a new resource, it should follow this format:

```
resource "aws_security_group" "allow_tls" {

    name = "allow_tls"

    description = "Allow TLS inbound traffic"

    vpc_id = aws_vpc.main.id

    ingress {

        description = "TLS from VPC"

        from_port = 443

        to_port = 443
        protocol = "tcp"
        cidr_blocks = [aws_vpc.main.cidr_block]
    }
    egress {
        from_port = 0
        to_port = 0
        protocol = "-1"
        cidr_blocks = ["0.0.0.0/0"]
        }
        tags = {
            Name = "allow_tls"
        }
}
```

When specifying the cidr_block to reference the EIP, enclose the ID in “${}”. Secondly, a subnet must be appended to the end of the quotes, in this case /32.

4.3 - Terraform Variables

When working with Terraform, there’s a significant chance that there will be multiple static values involved within the project e.g. ami, ports, commands etc. If any of these were to be changed, it will become very tedious to alter each value included and it's likely that one will be missed, leading to errors. To avoid this, the value can be stored in a variables file, which can then be referenced when required.

Variables are usually supplied from a file called variables.tf; a centralised location. When storing a variable, store in the format:

Variable “variable_id” {

    Default = “default_variable_value”

}

To reference a variable in a configuration file, in place of the value required, add:

[var.variable_id] including the parentheses.

Once done, when terraform apply is ran, the variables stored in variables.tf are referenced. This massively simplifies things in production for variable values subject to change, such as IP addresses, as the value needs to be altered just once.

4.4 - Variable Assignment

Variables can be assigned in terraform via four main methods:

- Environment variables
  - A fallback method in the event the others don't work
  - Terraform can search its environment to find a value to apply
  - To set an environment variable:
    - export TF_VAR_variable_id value
  - For this definition to take effect, must restart the terminal
- Command line flags
  - Not recommended approach unless altering one variable
  - During terraform plan or apply, append the following flag detailing the variable and value you want to apply:
    - Terraform plan -var="variable_id=value"
    - Usually used when want to quickly test the effects of a new variable
- From a file
  - Recommended over command line flags
  - In a new file "terraform.tfvars", can specify each variable's value in the form variable_id=2value"
    - Commonly used when wanting to set multiple variables simultaneously
  - To reference: terraform plan/apply -var-file="file_name.tfvars"
  - Note: no default value should be set in the variables.tf for this
- Variable defaults

    - Store variable values in variables.tf and specify default values, referencing via “${var.variable_id}”

It should be noted that if no additional variable values are specified, the value will be assumed to be the default value (specified in the configuration or variables tf file).

If a default value isn’t specified, you will be required to enter a value during the planning and execution phase.

4.5 - Data Types for Variables

When defining a variable, it’s good practice to define its data type during implementation, helping to prevent errors by restricting the type of value that will be accepted e.g. type = string.

Example:

Consider a company where every employee has a particular identification number, if that employee wanted to create a form of infrastructure, it should be done with that number only. So in variables.tf, the variable “instance_name” should be of type number.

Suppose that in the terraform.tfvars the value for instance_name is set to a value that isn’t of that data type/the data type also isn’t specified, eg. john-123; what will happen? This value will not be accepted and the plan will fail.

To specify a variable’s data type, simply add the type in the variable in variables.tf in the form type = type_attribute.

Key data types used include:

- String
    - Sequence of unicode characters representing text
- List
    - Sequential list of values identified by position within the list, position starting with 0 e.g. [“London”, “Paris”, “Helsinki”]
- Map
    - A group of values categorised by labels e.g. {name = “Nathan”. Age = 23}

- Number
    - Numerical values e.g. 42

By defining data types in variables.tf, users can use this as a reference point when defining variables in the tfvars file. In some cases if a variable type isn't specified, errors will arise as the program will assume a different type is expected.

4.6 - Fetching Data from Maps and Lists in Variables

When working with lists in terraform, sometimes you wish to reference a particular value from that list, rather than include all the values.

When referencing items from a map, follow the format: var.map_id["map_key"]

When referencing items from a list: var.list_id[list_position]

List positions ALWAYS start from [0].

4.7 - Count and Count Index

The count parameter on resources can simplify configurations and allow easier scalability of configurations.

Commonly if wanting to create a small number of the same resource e.g. 2 identical instances, could quickly just define them as separate instances. However, if want a larger amount of resources, can use count to save code space. Inside the resource, simply add count = value.

In resource blocks where count is set, an additional count object is available in expressions, so each instance's configurations can still be modified. This object has just one attribute: count.index, which starts with 0 for the first instance and continues like a list index. This is commonly used for altering properties such as the name.

When wanting to utilise the count index, append .${count.index} to the property chosen. For example this will make 3 machine instances of name:

- Machine_instance_name.0
- Machine_instance_name.1

- Machine_instance_name.2

So now the three instances can be uniquely identified. However, the above scenario isn't a commonly used naming convention, usually instances are configured for production processes like staging, development etc. Count index can still be applied here, but it will be referencing positions in a list.

To apply count index in this manner, apply the following to the field of choice: var.list_variable_id[count.index]. This will iteratively look through the list defined and apply each entry.

4.8 - Conditional Expression

Expressions using booleans to select one of two values, true or false. The syntax follows: condition ? true_val : false_val, with the appropriate value applied depending on the boolean result.

Example: Suppose there are two resource blocks as part of a configuration, depending on the variable value, only one of the resource blocks should run e.g. test and production resources.

For each resource, add a attribute property followed by:

attribute = var.test_variable_name == true ? true_value : false_value

And in the other dependent resource:

attribute = var.test_variable_name == false ? true_value : false_value

The boolean variable is defined as variable "variable_name" {}, with a default value set in terraform.tfvars

4.9 - Local Values

A local value assigns a name to an expression, allowing it to be used multiple times within a module without repeating it. This is commonly used for defining services, ownership names etc in the form of tags.

To define local tags, add: locals {}, then if you wish to categorise local values can add category1 = {}, category_n = {} within and so on. For example:

```
locals {

  common_tags = {

    Owner = "DevOps Team"

    service = "backend"

  }
```

To reference these values at any point, add local.category_name.

The local values have many use cases, a common one being conditional expressions, for example:

Name_prefix = " ${var.name != "" ? var.name : var.default}"

The above example defines a naming convention, if the value for var.name is left blank, it will be replaced by a default name.

In general, local values can be helpful to avoid repeating the same values or expressions multiple times.

However, they should be used in moderation, as they can make a configuration hard to read by future users of the files; they should only be used in situations where a single value or result is used in many places and that value is likely to be changed.

4.10 - Functions:

Terraform has many built-in functions that can be used to transform and combine values. The general syntax for a function is the function name followed by arguments separated by commas.

User-defined functions aren't supported. The categories of the functions are:

- Numeric
- String

- Collection
- Encoding
- Filesystem
- Date and Time
- Hash and Crypto
- IP Network
- Type Conversion

Further details for each function available on HashiCorp Website. To test functions, run the command terraform console; functions can then be added to the configuration.

A popular function is lookup, which can be used to look up the value of a single element from a map given its key; if the key doesn't exist, a default value is used.

lookup(map, key, default)

Another includes element: `element` retrieves a single element from a list in the form: `element(list, index)`

4.11 - Data Sources

Data sources allow data to be fetched or computed for use elsewhere within the terraform configuration. For example if an AWS EC2 instance was to be configured, the ami will differ depending on the region. If you set an ami specific to one region whilst your actual region is set to something different, the configuration will not be applied.

Rather than manually hard code the ami, it's possible to use it as a data source to filter the appropriate amis for the given region. Data source code is first defined under a "data" block and reads from a specific data source (Aws_ami) in this case, and exports it to app_ami.

```
data "aws_ami" "app_ami" {

 most_recent = true

 owners = ["amazon"]

 filter {
    name   = "name"
    values = ["amzn2-ami-hvm*"]
  }
}
```

Now when terraform plan is applied, the data source block will automatically search for the latest iteration for the amazon linux 2 ami for the chosen region. This can be altered for different owners etc.

4.12 - Terraform Debugging

Terraform tracks all changes in a series of logs; which can be enabled by setting the TF_LOG environment variable to any value. It can be set to TRACE, DEBUG, INFO, WARN or ERROR to change the overview of the log's display.

To do so, run: export TF_LOG=value

Then apply chosen commands e.g. terraform plan.

To save the logs, can run export TF_LOG_PATH=/path/to/log/terraform-crash.log

Now when a command is ran, all logs will be added to the above file.

TRACE is the most extensive overview for TF_LOG and is the default setting assuming it's set to something other than a log name.

To persist a logged output, define a TF_LOG_PATH to force the log to append to a specific file.

4.13 - Terraform Format

When working with Terraform, readability is very important. A common good practice is to ensure that all equals signs are aligned for key value pairs. To format in this manner, run the command terraform fmt. This will automatically apply any indentations and alignments necessary to make the file easier to read.

4.14 - Validate Config Files

Prior to running terraform plan, it's important to ensure that your configuration files are syntactically correct. Otherwise, when plan and apply are ran, errors may occur which can derail things. To validate, run terraform validate. This will check the syntax for the configuration files, ensuring there are no incorrect attributes, all variables are declared etc.

4.15 - Load Order and Semantics

Generally, terraform will load all the configuration files within the specific directory in alphabetical order; as long as the files end in .tf

In general practice, code should be split into multiple files. For example a file for providers, variables and one for each resource(s) to be created. This allows easier management of infrastructure e.g. can just go to 1 file to add a new instance or a user etc.

Note, when adding two of the same resource, you must give different IDs after defining the resource type.

4.16 - Dynamic Blocks

In many cases, there are repeatable nested blocks of resource code that need to be defined. If not managed carefully, this could lead to long stretches of code that are difficult to manage. Commonly this occurs with security groups, defining ingress, egress rules etc.

To get around this, can use a dynamic block, specified by the "dynamic" before the property specifier, from which the dynamic block allows you to iteratively add content defined in a separate variable list or map. Considering an ingress property:

```
resource "aws_security_group" "dynamicsg" {

  name        = "dynamic-sg"

  description = "Ingress for Vault"



  dynamic "ingress" {
    for_each = var.sg_ports

    iterator = port

    content {

      from_port   = port.value

      to_port     = port.value

      protocol    = "tcp"

      cidr_blocks = ["0.0.0.0/0"]

    }

  }


  dynamic "egress" {

    for_each = var.sg_ports

    content {

      from_port   = egress.value

      to_port     = egress.value

      protocol    = "tcp"

      cidr_blocks = ["0.0.0.0/0"]

        }

  }

}
```

This dynamic block will iteratively fill out the contents of the ingress and egress blocks with content defined in a separate variable as a list. The egress dynamic block is then filled iteratively based on the ingress block.

Iterators are optional arguments which set the name of a temporary variable that represents the current element of a more complex value.

If omitted, the name of the variable defaults to the dynamic block's label, as shown in the egress block; iterators are commonly used to improve readability.

4.17 - Tainting Resources

Consider a scenario where a new resource has been created, but users have made a lot of manual changes to it in terms of both infrastructure and within the server. To deal with this, can either import the changes to terraform or delete and recreate the resource to update the configuration.

Terraform taint command manually marks a terraform resource as "tainted", this forces the resource to be destroyed and recreated the next time terraform apply is run.

To taint, run the command: terraform taint resource_type.resource_id

4.18 - Splat Expressions

Splat is an expression that produces a list of all the attributes, denoted by *.

Splat essentially denotes, "anything that fits with the value I've just entered", a detailed example follows:

```
#AWS Provider Configuration

provider "aws" {

  region     = "eu-west-2"

  access_key = "AKIAIDHLXIC5PHKA4HRA"

  secret_key = "N1y+4EUvS7yvgdvQCyZM60To1LPOlePVC9dd1CJD"

}



resource "aws_iam_user" "lb" {
    name = "iamuser.${count.index}"

    count=3

    path = "/system/"

}


output "arns" {

    value = aws_iam_user.lb[*].arn

}
```

The above configuration looks to create 3 iam users in aws. The focus lies in the outputs however; specifically the value. The value aws_iam_user.lb[*].arn will look for all values like this and display them, so the resultant output is details the 3 iam users parsed into their constituent parts:

```
arns = [

  "arn:aws:iam::746085785702:user/system/iamuser.0",

  "arn:aws:iam::746085785702:user/system/iamuser.1",

  "arn:aws:iam::746085785702:user/system/iamuser.2",
```

This could also be applied to any other listable properties; officially: A *splat expression* provides a more concise way to express a common operation that could otherwise be performed with a `for` expression.

4.19 - Terraform Graph

A command used to generate a visual representation of a configuration or execution plan. Expressed in the DOT format; which can be easily converted to an image. This is a great way to visualise resource dependencies.

To run: terraform graph > filename.dot

The dot file is expressed as a text, to change to an image, can use a graph visualisation package such as graphviz.

4.20 - Saving Terraform Plan to a File

When generating a terraform plan, it can be beneficial to save it to a particular path. By doing so, this plan can be viewed later or can be used to apply changes specified only by that plan.

To save a plan to a file: terraform plan -out=filename

To apply a plan saved to a particular file: terraform apply filename

4.21 - Terraform Output

A command used to extract the value of an output variable from the state file

Configured by: terraform output state_file_attribute

Can be good to use for verification and debugging purposes. The same outputs could be manually coded in the configuration file or inspecting the terraform state file

5.0 - Terraform Provisioners

5.1 - Introduction to Provisioners

Prior to this section had only been focused on creation and destruction of infrastructure, in practice we want to do something with that infrastructure, such as run particular software. To install this software and run commands, we use provisioners.

Provisioners are used to execute scripts on a local or remote machine during resource creation or destruction. A classic example is installing NGINX Web server on a Web-Server VM.

```
provisioner "remote-exec" {

   inline = [

     "sudo amazon-linux-extras install -y nginx1.12",
    "sudo systemctl start nginx"
  ]

  connection {
    type        = "ssh"
    user        = "ec2-user"
    private_key = file("~/.ssh/authorized_keys/testprovsionerEC2.pem")
    host        = self.public_ip
  }
}
```

When defining the remote exec, must define the inline commands to be ran as well as the method of connection with the appropriate authentication means.

5.2 - Types of Provisioners

There are 2 main types of provisioners:

- Local-exec
  - Allows invoking of local executables after the resource is created
  - Commands defined ran on the machine where terraform apply was ran i.e. the local machine
- Remote-Exec
  - Allows invoking of commands and scripts directly on the remote machine

Others are available, but the above 2 are the most widely used.

5.3 - Remote-Exec Implementation

For remote-exec to run, the resource must be created first i.e. make sure the resource block ‘s main attributes are added before defining the provisioner.

Within the provsioner section, must define 2 properties: inline and connection.

```
resource "aws_instance" "myec2" {

  ami           = "ami-0a13d44dccf1f5cf6"

  instance_type = "t2.micro"

  key_name      = "remote-exec-keypair"

  #configure provisioner with inline commands

  provisioner "remote-exec" {

    inline = [

      "sudo amazon-linux-extras install -y nginx1.12", #install nginx

      "sudo systemctl start nginx"                     # start nginx

    ]

    connection {

      #connection method

      type = "ssh"

      user = "ec2-user"

      #private key for authentication

      private_key = file("./remote-exec-keypair.pem")

      host        = self.public_ip

    }

  }

}
```

In the “inline” block, any commands you wish to run in the remote machine must be added here, separated by a comma. Take care to ensure that the commands are syntactically correct for operating system the VM or instance will run.

As well as the inline block, you must also specify the attributes associated with the connection to the VM to run the commands. First the type, then the user, which can be named however you want, then the private key and the host, in this case the machine itself.

Authentication with the private keys is where it gets fiddly:

- Create a key pair on the aws console and download the .pem file to the directory containing the configuration files or a location of your choice
- Use the file function to specify the key file location as the value for the private_key attribute: file("/path/to/file.pem")
- In the resource block, before the "provisioner" section, add key_name = key_pair name as specified in the console

5.4 - Local-Exec Implementation

Local-exec provisioners allow commands to be run on the local machine immediately after a resource is created. One of the most common uses is running ansible-playbooks on the created server after the resource is created.

Defined in a very similar manner to remote-exec; local-exec. Specify commands via command = define_command. There are no connections to be made so there is no need to specify the connection parameters.

6.0 - Terraform Modules and Workspaces

6.1 - DRY Principle

A principle specifically focused on the aspect of software engineering. DRY or Don't Repeat Yourself. The idea behind the principle is to reduce the repetition of software patterns. Previously, we were making static content into variables so that there can be a single source of information.

Another repeated form of code is the resource "aws_instance" which has been used across nearly every project. To make it easier we can centralize terraform resources and call out from them whenever required; this works in tandem with the practice that each resource is found within its own file.

To use a module to reference a repeatable resource block, adhere to the following structure:

Module “resource_id” {

    Source = “/relpath/to/file”

}

So when terraform plan, apply etc is run, the configuration looks for the corresponding module file to obtain the resource details. It should be noted that the filename in the source quotes shouldn’t include the .tf extension; terraform knows what to look for.

6.2 - Module Implementation: EC2 Instance

Adopting the following architecture:

- Root
  - Modules
  - Projects
    - A
    - B

The following can be added to a file in the modules folder:

```
resource "aws_instance" "myec2" {
  ami           = "ami-0a13d44dccf1f5cf6"
  instance_type = "t2.micro"
}
```

This defines the ec2 instance we’d like to create and reference in the projects folder.

Adding the provider information to a providers.tf, to begin using as usual must run terraform init. This time as well as the provider plugins, terraform will initialise the modules provided.

When the plan and application phases are executed, the resource module will be referenced in the project folder. The use of modules makes things significantly easier for management and readability, especially if the people working on the project aren't strictly familiar with terraform and the associated properties; they only need to reference it.

6.3 - Variables & Terraform Modules

In practice, a common challenge with infrastructure management is the need to build environments such as dev, staging and production with a generally similar setup, but with some different environmental variables.

Module variables cannot be overridden, if you wish to change the values of a property, you can create a variables.tf file in the modules folder to reference. As before to reference the value, use **var.variable_id.**

Now if a value is hardcoded along with the source module, the variable value can be overwrote from the default. Permissions can be set to prevent unnecessary overwriting however.

6.4 - Terraform Registry

A repository of modules written by the terraform community; it can help getting started with terraform more easily. Included within are verified modules that are maintained by 3rd party providers, as well as community submitted providers.

Verified modules are reviewed by HashiCorp and are actively maintained to remain up to date with terraform and their respective providers.

In the terraform registry, you are provided with usage examples, a tutorial on how to use the module and any associated inputs and what they mean.

To use a terraform registry module within the code, you can make use of the source argument, detailing the module path in the form:

Source = "terraform/path/to/module"

And specify any other parameters required, you can hard code version for compatibility purposes. This method doesn't require a separate module folder as just using modules from the registry.

6.5 - Terraform Workspace

Workspaces are groupings of application windows used by a window manager.

Terraform allows the use of multiple workspaces with each able to use a different set of environment variables. This is often useful for running production and staging environments on the same machine.

To use the difference in workspaces, utilise terraform workspace:

Command	Definition
Terraform workspace list	List the existing workspaces Current workspace denoted by *
Terraform workspace select workspace_name	Switch to pre-existing workspace
Terraform workspace new workspace_name	Create a new workspace (switches automatically)
Terraform workspace delete workspace_name	Delete a particular workspace; use -force to force deletion if not an empty workspace
Terraform workspace show	Show current workspace

6.6 - Implementing Terraform Workspace

Consider a scenario where for a different workspace you have a variable that's subject to change, for example the instance_type. If in the default workspace, you want it to be size 1, size 2 for staging and so on. To utilise this, we can use workspaces in tandem with variables.

Defining the variable "instance_type" as a map:

Variable "instance_type" {

```
        Type = “map”

        Default = {

                Workspace_1 = “size1”

                Workspace_2 = “size 2”

                Workspace_3 = “size 3”

        }

}
```

To reference this, as before can write var._variable_id with an added extra:

```
Instance_type = lookup(var.instance_type, terraform.workspace)
```

This looks to find the label in the instance_type map matching that of the current workspace.

Each workspace’s state file is stored in a new folder terraform.tfstate.d; the default workspace’s state file remains in the root directory.

7.0 - Remote State Management

7.1 - Integrating with Git for Team Management

Up until now any changes and configurations have been made locally, in practice this isn’t recommended as there is always the possibility that hardware failure may occur and the working method isn’t collaborative. There should be a centralized repository for the team to access and make relevant changes.

- Ensure repository account is set up e.g. GitHub
- Install Git CLI/GIT Bash
- Clone new repository into new directory
- Create files chosen
- Run git add . to add all files in directory

- Configure credentials:
    - Git config --global user.email “email”
    - Git config --global user.name “username”
- Commit changes:
    - Git commit -m “commit messages”
- Push changes: git push origin master

Note: When committing data to the git repository, AVOID pushing access and secret keys with the code for security purposes.

7.2 - Security Challenges in Committing TFState to GIT

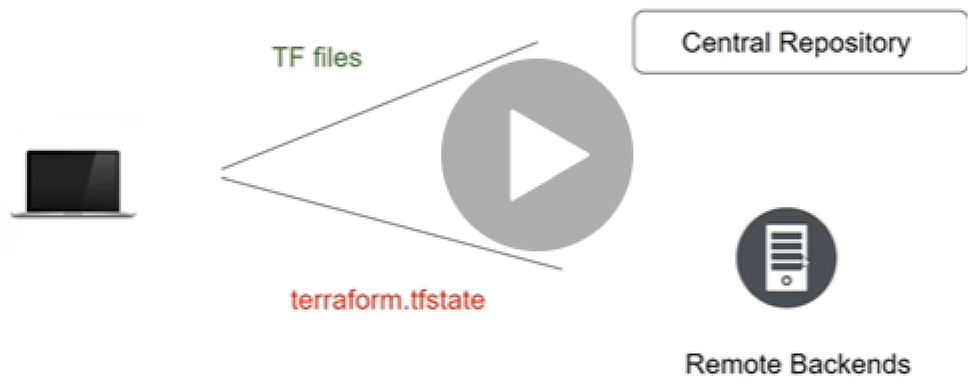
As mentioned in the previous section, for security purposes sensitive information such as usernames and passwords shouldn’t be stored in an online repository. For sensitive information, one could store it in separate files outside the repository to be referenced using the file function.

In practice this does work, however the password is then stored in the tfstate file which is then accessible should it be committed to the repository, so we’re back to square one. We must look to move the tfstate file out of the repository, but still allow the team to access it.

7.3 - Remote State Management

Remote BackEnd is a feature of terraform that allows you to store tfstate files in a central repository that isn’t easily accessible like Git.

The architecture follows:



One workaround is to store the tfstate file into a remote backend, such as an S3 Bucket in AWS.

Terraform supports various types of remote backends which can be used to store data; in general they can be split into 2 types:

- Standard BackEnd - Allows State Storage and Locking
- Enhanced BackEnd - All features of standard and remote management

7.4 - Implementing S3 Backend

Terraform has many different types of backends that can be used to store the tfstate file to go alongside different providers. For the purposes of AWS, we will create an S3 bucket.

Unless a remote backend is specified, by default the TFstate file will be stored locally.

To implement a backend, create the following configuration file:

```
Terraform {

        Backend “s3” {
                Bucket = “bucket-name”

                Key      = “tfstate_filename.tfstate”

                Region = “region-of-choice”

        }

}
```

This in theory should work, however there has been a bug cropping up where the authentication keys cannot be located even if they’re defined in the providers.tf file. To fix, can add the access_key and secret_key to the above configuration.

Note: The S3 Bucket will have to be created manually on the AWS site for this to work.

Note: In the event of errors with initialization, delete the .terraform folder and try again.

7.5 - Challenges with State File Locking

Whenever a write operation is being performed, terraform locks the state file. This is hugely important as if someone tries to perform an action that would alter the state file whilst you are trying to do another alteration, the state file will become corrupted. A typical example would be:

- Person A terminating a resource associated with a tfstate file
- Person B tries to resize the resource during termination

State file locking automatically occurs when working with terraform in the CLI and the state file is stored locally. If the state file is stored in a remote backed, other methods are required.

7.6 - Integrating DynamoDB with S3 for state locking

To implement state locking on a tfstate file stored in an S3 bucket, caN utilise a dynamodb table. This DB table can be created via the AWS website.

To utilise, add a property to the backend resource titled dynamodb_table with the value being the table name.

To create a dynamoDB for locking, create the following resource block:

```
resource "aws_dynamodb_table" "terraform_state_lock" {
 name          = "terraform-lock"
 read_capacity  = 5
 write_capacity = 5
 hash_key       = "LockID"
attribute {
   name = "LockID"
   type = "S"
 }
}
```

7.7 - Terraform State Management

As the terraform usage becomes more advanced, there are some cases where the terraform state needs to be modified.

It's good practice to never modify the state file directly, instead make use of the terraform state command.

State Subcommand	Description
list	List resources in the state
mv	Move an item in the state
pull	Pull current state and output to a standard output form
push	Update remote state from a local state file
replace-provider	Replace a provider in the state
rm	Remove instances from the state
show	Show a resource in the state

The mv command is used in many cases when you want to rename an existing resource without destroying and recreating it. The command will output a backup copy of the state prior to saving any changes.

The mv command's syntax follows the form:

Terraform state mv [options] /path/to/source

OR

Terraform state mv 'name1' 'name2'

The pull command is used to manually download and output the state from a remote state. This is useful for reading values out of the state file, possibly pairing it with other relevant commands.

The push command, whilst rarely used, is used to manually upload a local state file to a remote state.

The rm command can be used to remove items from the state, this doesn't destroy it! Any items removed from the state file will continue to run; they simply aren't managed by terraform anymore.

The show command is used to show all the attributes associated with a single resource, the command is: terraform state show resource_type.resource_id.

7.8 - Importing Existing Resources with Terraform Import

In the event that a resource has already been created via manual methods, any changes to be made must be made manually. To do so, can utilise the terraform import command.

Terraform is able to import existing infrastructure. This allows you to take resources you've created by some other means and bring it under Terraform management.

To import a resource, such as an existing ec2 instance, first write a resource block for it in your configuration, establishing the name by which it will be known to Terraform:

```
resource "aws_instance" "example" {

  # ...instance configuration...
}
```

The resource block could be filled with the various attributes associated with the instance, however it’s acceptable to leave it blank until the resource is imported.

To import, run the command: terraform import aws_instance.example i-12345

This will look up the AWS resource (instance in this case) with ID i-12345 and attach the existing settings of the instance as described by the AWS console to a module titled aws_resource.resource_ID and save the mapping to the terraform state.

As a result, any changes you would like to make to the resource must now be done via terraform, you can see how the configuration compares to the imported resource using terraform plan.

8.0 - Security Primer

8.1 - Handling Access and Secret Keys

Currently we stored the authentication keys for AWS in the providers.tf file. This is bad practice as if this file could be used to access the keys and from this access the AWS resources. To avoid this, download the AWS CLI and run AWS configure.

By running aws configure, you configure the CLI to match with the AWS account being used, as you provide the secret and access keys as part of this command; as well as the region.

8.2 - Provider Use Case: Resources in Multiple Regions

In some scenarios, it's possible that you would wish to configure the provisioning of resources to multiple regions. This can be challenging as it can involve multiple sets of authentication keys as well as different regions.

Prior to this section, the aws-region parameter had been hard-coded into the providers.tf file. If this parameter is removed, terraform will request you specify the region. In some cases you may wish to deploy multiple instances to different regions on one account or deploy instances to multiple accounts.

Considering the first case, where we wish to deploy to different regions on one account. We can make use of an alias variable. This is because you are not allowed to have multiple provider configurations unless they can be differentiated. To differentiate, introduce the alias variable. This is essentially a way of adding an ID variable to each provider.

```
provider "aws" {

  region     =  "us-west-1"
}
```

```
provider "aws" {
  alias      =  "aws02"
  region     =  "ap-south-1"
  profile    =  "account02"
}
```

Now if you wish to deploy a particular resource to a region with the alias, you must specify the provider for that resource by the following attributes:

Provider = “provider_type.provider_alias”

In this case: provider = “aws.aws02”

8.3 - Handling Multiple AWS Profiles with Providers

When considering deploying resources to multiple accounts, you want to work with the multiple aws accounts, you need to make use of the aws credentials file configured during the aws CLI installation; located at ***"%USERPROFILE%\.aws\credentials"*** on windows.

The credentials file follows the form for multiple accounts:

[account01]

aws_access_key_id = ACCESS_KEY

aws_secret_access_key = SECRET_KEY

[account02]

aws_access_key_id = ACCESS_KEY

aws_secret_access_key = SECRET_KEY

To utilise the credentials from a specific account, in the provider chosen, add the following:

Profile = “account_name”

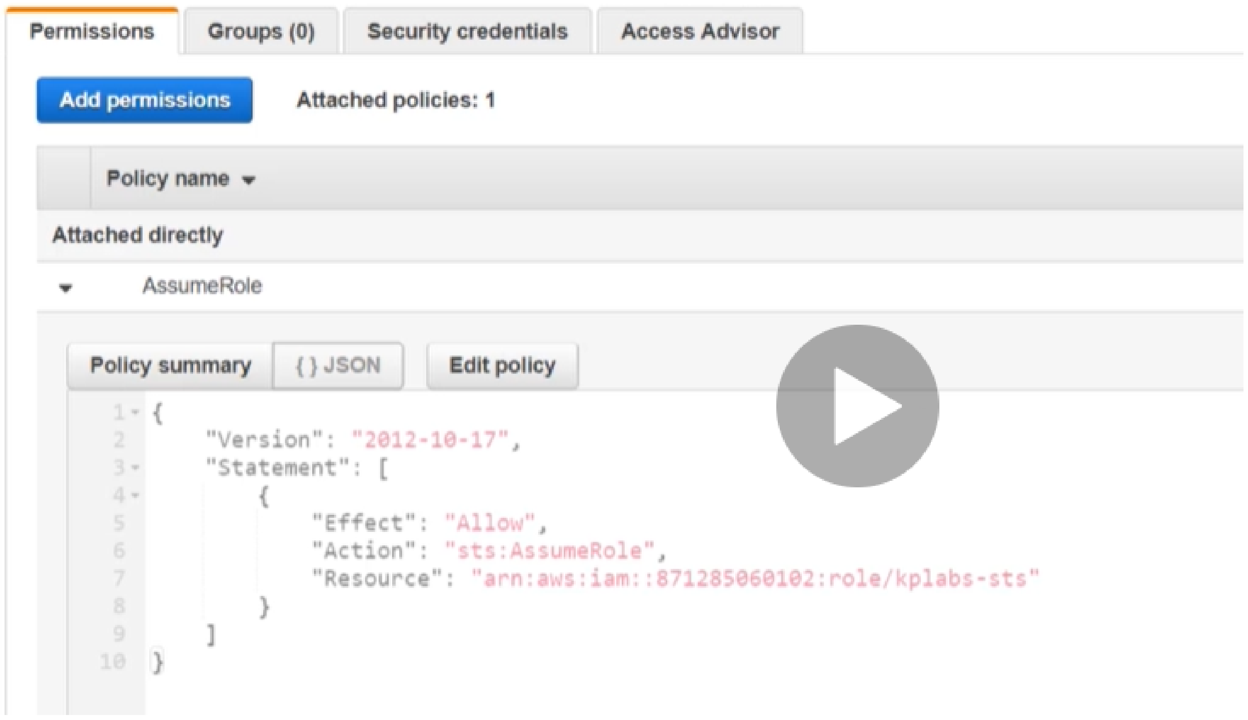
This will tell terraform to look in the credentials file to identify the credentials for the account “account_name” and deploy any resources using that provider configuration to that particular account.

8.4 - Terraform & Assume Role with AWS STS

When working with multiple accounts and their associated credentials, it’s advisable to make use of the Assume Role functionality of the AWS Security Token Service (STS) This is a web service that enables you to request temporary, limited-privilege credentials for AWS Identity and Access Management (IAM) users or for users that you authenticate (federated users).

By using this, you can keep a single set of usernames and password, along with authentication keys on an IDENTITY ACCOUNT; each account underneath can then be accessed by using ASSUME ROLE.

Assume role can be set for IAM user permissions:



```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "sts:AssumeRole",
            "Resource": "arn:aws:iam::871285060102:role/kplabs-sts"
        }
    ]
}
```

To provision resources via an account that has assume role privileges:

Aws sts assume-role --role-arn arn:url --role-session-name session_name

To provision resources via assume-role, need to add particular values to the provider.tf file to specify that the actions are to be done under an assumed role:

Within the provider resource block add:

Assume_role {

    Role_arn = “find_role_arn”

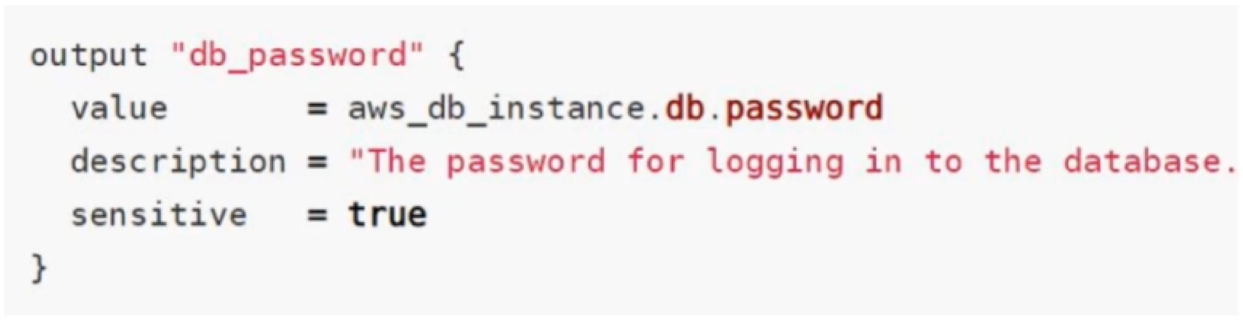
    Session_name = “session_name”

}

Role ARNS are resource-specific, session-name can be chosen to the user’s desire.

8.5 - The Sensitive Parameter

When managing large sets of infrastructure in terraform, it’s likely that you will see some sensitive information involved such as passwords. When working with fields containing information of this nature, it’s best to use the sensitive property on it; setting it to true.



```
output "db_password" {
  value       = aws_db_instance.db.password
  description = "The password for logging in to the database.
  sensitive   = true
}
```

By setting the sensitive parameter to “true”, the field’s value will be prevented from being displayed in the CLI and in terraform cloud; it will not be encrypted in the state file however.

9.0 - Terraform Cloud and Enterprise Capabilities

9.1 - Terraform Cloud Overview

Terraform Cloud manages Terraform runs in a consistent and reliable environment. It provides various features such as access controls, private registry for sharing modules, policy controls and more.

Terraform Cloud Projects are stored in workspace repositories. Inside these, information detailing the project can be found along with additional info regarding Terraform runs, such as plan details, estimated monthly costs for resources; and Terraform apply details.

In some cases Policy checks may be present, this essentially is to verify any tags associated with resources. This is often necessary for identification purposes.

Users are allowed to comment on runs to keep track of progress and provide updates when necessary.

It's also possible to set environment variables in Terraform Cloud as well as viewing the tfstate file associated with the project.

Finally, Terraform Cloud can be linked with Github repositories for projects, so when any changes are made to that repo, they are subsequently made to the terraform workspace.

9.2 - Creating Infrastructure with Terraform Cloud

Terraform's pricing depends on the user requirements, for teams and governance more features are required; subsequently increasing price. To create an account go to https://app.terraform.io/signup/account.

When getting started, you must first create an organisation to create a workspace; simply choose a unique organisation name and provide the email address you wish to associate with it.

Next, link your chosen form of version control, usually this will be your GitHub account.

In terraform cloud, navigate to: Settings -> VCS Providers and select add new provider; select as appropriate.

Suppose GitHub is the VCS Provider we wish to connect with, the following needs to be added:

- An optional display name for the VCS Provider (usually needed if configuring multiple instances of the same provider)
- Client ID
- Client Secret

To add the latter two fields, some setup via GitHub is required. In GitHub, create a repository that you wish to link if one doesn't already exist. From here navigate to settings -> developer settings -> oauth applications.

From here can register a new OAuth application, detailing the application name, home page url (see terraform documentation), and a template callback url to be replaced. Once the link is created, you will be presented with the client ID and secret to be added to the Terraform Cloud VCS Provider setup.

Adding the VCS Provider will create the callback url which can then be added to the github application; finally the applications can be connected.

To start creating infrastructure with Terraform Cloud, first add/commit some file(s) to the GitHub Repository, then create a workspace in Terraform Cloud. When creating a workspace, connect to the chosen VCS provider and select the desired repository.

Once the configuration is complete, terraform variables and environmental variables must be configured. This can include ACCESS KEYS for AWS, default region etc.

From here you can queue a plan, which will initiate a terraform plan using the code linked in the repository.

If the plan is successful, the terraform apply command can be ran, or a comment can be added alongside, this can be useful for troubleshooting and debugging during production.

It should be noted that the terraform state file is stored on Terraform Cloud, plan and infrastructure destruction capabilities are also available.

For production environments or plans, it is possible to show the cost estimation for running projects and applying configurations.

9.3 - Sentinel

Sentinel is an embedded policy-as-code framework integrated with the products provided by HashiCorp. It allows fine-grained, logic-based policy decisions; which can be extended to use info from external sources. It's a paid feature of terraform and acts as an inbetween for terraform plan and apply stages.

In a hierarchical view, a sentinel policy would be put in place e.g. refusing to accept an EC2 resource without tags being applied, the policy would be attached to a policy set which can then be applied to a workspace.

To create a policy set:

- Navigate to settings -> policy set -> connect a policy set
- Configure VCS connection as necessary
- Configure settings for policy, choose to apply to workspace(s)

To create the policy:

- Settings -> policies -> create policy
- Add details where required
- Set enforcement mode:
    - Hard-Mandatory: Cannot override
    - Soft-Mandatory: Can be overridden
    - Advisory: For logging purposes only
- Add policy code (See terraform documentation)
- Associate the policy with a policy set

Now when a plan is queued, should it be successful the policies will be checked to determine if an apply can be ran with appropriate logs.

Example code:

```
import "tfplan"

main = rule {
 all tfplan.resources.aws_instance as _, instances {
   all instances as _, r {
     (length(r.applied.tags) else 0) > 0
   }
 }
}
```

9.4 - Remote Backend Overview

The remote backend stores Terraform state files and may be used to run operations in the Terraform Cloud. Terraform Cloud may also be used with local operations, in which case only the state is stored in the TFCloud Backend.

Remote Operations:

When using full remote operations, commands like terraform plan can be executed in TFCloud's runtime environment, with log output streamed to the local terminal.

To configure the backend, the following must be applied to the terraform configuration files:

- In the file containing the resource(s), add a terraform block containing "backend "remote" {}" alongside the required version if necessary
- In a file titled backend.hcl
  - Workspaces { name = "repository_name" }
  - Hostname = "app.terraform.io"
  - Organisation = "organization_name" (tfcloud organisation name)

Once setup, when terraform plan or apply is ran, it will be run in the browser/online, the logs can be viewed via this method. Additionally, cost estimations and sentinel policies will be checked if they're in place.

It should be noted that if resources are configured locally and are wanting to use remote operations, you cannot use a workspace that uses a VCS connection.

9.5 - Implementing Remote Backend Operations

1. Create workspace without VCS Connection
2. Configure backend.hcl file, detailing workspace, hostname and organisation info
3. Configure resource configuration files with terraform block containing backend "remote" {}
4. To initialise with backend file: terraform init -backend-config=backend.hcl

For authenticating with a remote backend, need to create a token. To do so, run terraform login, which stores the credentials to a particular path once successful. From here the API token can be found on the TFCloud website, which is to be copied into the user input requested.

Once terraform is logged in, step 4 can be re-ran, ensure that any necessary environment variables are set on the TFCloud workspace.

10.0 - Exam preparation

10.1 - Important Pointers

Overview

Exam Property	Description
Type	Multiple Choice
Format	Online Proctored (Overviewed by an instructor/vigilator)

Duration	1 Hour
Question Number	57
Price	70.50 USD (Tax Not Included)
Language	English
Expiration Time	2 years

Question types:

- True/False - Statement provided, answer if true or false
- Multiple Choice - Given a piece of code, what code should be applied into area / given a current scenario, what answers are applicable
- Text Match - Given a text box, what code would have to be added

Example Questions:

1. What is the name of the file that stores state information?
    a. **terraform.tfstate**
2. When referencing a file, for terraform 0.11, it would follow a format similar to "${file("path/to/file")}", how does this translate to terraform v0.12?
    a. **file("path/to/file")**
3. What does terraform init do? (Could expect multiple options)
    a. Definition files can be a bit ambiguous, choose most appropriate

As the exam is online proctored, need the following:

- Zoom is required
- Alone
- Desk and work area are clear
- Power source is applied
- No phones or headphones
- No dual monitors
- No leaving seat
- No talking

- Webcam, speakers and mic must remain on
- Proctor must be able to see you

Must show the entire room and desk before the exam can begin.

Note: May be asked to move certain aspects or adjust settings.

Once booked/register, make sure to review system requirements and start approx 15 minutes before the appointment.

Important Pointers

Providers

- A provider is responsible for understanding API interactions and exposing resources
- Most providers correspond to one cloud or on-premises infrastructure platform, offering resource types corresponding to each of the features on that platform
- You can specify the provider version in the provider block
- Alternatively use terraform init -upgrade to automatically upgrade to the latest acceptable version of the chosen provider(s)

Provider Architecture

Terraform Providers act as the link between actual cloud providers and terraform configurations. They are responsible for the underlying API interactions and authentication between your machine and the provider

It is possible to use multiple providers in a configuration via the use of “alias”, an attribute added to the provider block to distinguish itself from the default (whichever provider you choose to not provide an alias for).

Terraform Init

- Used to initialize the working directory containing Terraform configuration files.
- The configuration is searched for module blocks, so the source code for referenced modules is retrieved from the locations given in the source arguments
- Any providers specified must be initialised by terraform before it’s to be used
- No additional files are created with Terraform Init

Terraform Plan

- Creates an execution plan of the infrastructure described in configuration files
- Doesn't modify the infrastructure in any form
- Performs a refresh unless told otherwise, provides a plan detailing the necessary actions to achieve the desired state in the configuration files
- This is a convenient way to check the execution plan matches expectations without applying changes

Terraform Apply

- Used to apply changes required to reach the desired state of the configuration
- Writes data to the terraform.tfstate to detail changes
- Once apply is complete, resources are immediately available

Terraform Refresh

- Used to refresh the state file for the configuration to remind the user of any changes or ensure any changes have been made
- Doesn't modify infrastructure, but does modify the state file

Terraform Destroy

- Used to destroy any terraform-managed infrastructure
- Not the only method of infrastructure destruction, could also remove desired resources from configuration files, terraform apply will then look to destroy those resources

Terraform Format

- Used to rewrite Terraform Configuration files to a canonical format and style
- Recommended for use when all configuration written by team members must adhere to a well-written style

Terraform Validate

- Validates the configuration files in a directory
- Checks whether a configuration is syntactically valid

- Useful for general verification of reusable modules as well as attribute names and value types
- Generally run automatically as a test step or post-save check for a reusable module in a CI system prior to Terraform plan
- Requires an initialised working directory with all plugins and modules required installed

Terraform Provisioners

- Used to model specific actions on a local machine or a remote machine to prepare servers or other forms of infrastructure for service
- Should only be used as a last resort, such as using Packer in conjunction with Terraform.
- Provisioners are located within resource blocks.
- For remote-exec, the inline commands must be defined as well as the connection method in a similar manner to below:

```
provisioner "remote-exec" {

   inline = [

     "sudo amazon-linux-extras install -y nginx1.12", #install nginx

     "sudo systemctl start nginx"                     # start nginx

   ]

   connection {

     #connection method

     type = "ssh"

     user = "ec2-user"

     #private key for authentication

     private_key = file("./remote-exec-keypair.pem")

     host        = self.public_ip

   }
```

```
  }
```

For local-exec, a similar manner follows, without the need for connections to be defined, commands are defined by the “command” attribute rather than “inline”:

```
provisioner "local-exec" {
    command = "echo ${aws_instance.myec2.private_ip} >> privateips.txt"
  }
```

Debugging Terraform

- Terraform has logs that can be enabled by setting the TF_LOG environment variable to a value of choice, determining the verbosity of the logs.
- The value of TF_LOG may be any of:
    - TRACE
    - DEBUG
    - INFO
    - WARN
    - ERROR
- To persist logged output, set TF_LOG_PATH=”/path/to/file”
- Log file saved as .log format

Terraform Import

- Terraform can import existing infrastructure
- Allows taking resources created via other means and make it manageable by Terraform
- Current implementation can only import resources into the state, it doesn’t generate configuration
- Before running terraform import, write a resource configuration block manually for the resource, which the imported object can be mapped to
- E.g. terraform import aws_instance.myec2 instance-id

Local Values

- Assigns a name to an expression, allowing easy reuse within a module without manually typing code out
- A local value expression can refer to other locals, but as usual reference cycles aren't allowed. A local cannot refer to itself or to a variable that refers back to it
- It's recommended to group together logically-related local values to a single block, especially if they depend on one another

Data Types

- There are 4 main data types involved with Terraform:
  - String - Unicode characters representing text
  - List - Sequential list of values identified by their position, first value being position "0"
  - Map - A group of values identified by named labels/keys e.g. age=52
  - Number - Numerical values

Terraform Workspace

- Terraform allows multiple workspaces
- Each workspace can have a different set of environmental variables associated
- Additionally, this allows multiple state files for the same configuration, can be beneficial for testing and production environments
- Additional workspace state files stored in terraform.tfstate.d folder
- Default workspace state file stored in root directory

Terraform Modules

- Terraform resources can be centralised and called out from TF module files when required

Terraform Modules - ROOT and Child

- Each terraform configuration has at least one module, the root module, which consists of the resources defined in the .tf files in the main working directory

- Modules can call other modules, which allows the inclusion of the child module's resources into the configuration in a concise manner
- A module that includes a module block in the following manner is a child module:

```
module "myec2" {

    source = "../../modules/ec2"

}
```

Modules - Accessing Output Values

- The resources defined in a module are encapsulated, the calling module cannot access their attributes directly
- The child module can declare output values to selectively export certain values to be accessed by the calling module in a similar manner to:

```
#Configure aws bucket output

output "mys3bucket" {

  value = aws_s3_bucket.mys3.bucket_domain_name
```

Suppressing Values in CLI Output

- An output can be marked as sensitive information using an optional "sensitive" argument in a similar manner to:

```
#Configure aws bucket output

output "db_password" {

  value           = aws_db_instance.db_password

  Description     = "The password for logging into the database"

  Sensitive       = true

}
```

- Setting an output value in the root module as sensitive prevents Terraform from showing its value when output after Terraform apply

- Sensitive outputs are still recorded in the state as fully accessible values, anyone who can access the state file can view the sensitive information

Module Versions

- It's recommended to constrain the acceptable version numbers for each external module to avoid unexpected changes.
- Version constraints are supported for modules installed from a module registry e.g. Terraform Registry or Terraform Cloud's private module registry in a similar manner to:

```
module "ec2_cluster" {
  source                 = "terraform-aws-modules/ec2-instance/aws"
  version                = "~> 2.0"

  name                   = "my-cluster"
  instance_count         = 1

  ami                    = "ami-0a13d44dccf1f5cf6"
  instance_type          = "t2.micro"

  subnet_id              = "subnet-5fbf1013"

  tags = {
    Terraform   = "true"
    Environment = "dev"
  }
}
```

Terraform Registry

A registry directly integrated into terraform, containing modules submitted by the terraform community, including 3rd party providers

- The syntax for referencing a registry module is:
- <NAMESPACE>/<NAME>/<PROVIDER>

Private Registry for Module Sources

- You can also use modules from a private registry, such as those provided by Terraform Cloud
- Private registry modules have source names of the format:
- <HOSTNAME>/<NAMESPACE>/<NAME>/<PROVIDER>
- Note: This is the same as that of a public registry module with the prefix detailing the hostname.
- When fetching a private registry module, a version must be specified.

Functions

- Terraform includes a number of built-in functions that can be used to transform and combine values
- No user-defined functions are supported, only built-in functions such as:
    - Min
    - Max
    - Element
    - Lookup

Count and Count Index

- The count parameter on resources can simplify configurations and let you scale resources by incrementing a number
- In resource blocks where count is set, an additional count object, count.index is available in expressions so the configuration of each resource can be modified e.g.

```
Resource 1 Configuration (Instance 1)

#resource "aws_instance" "instance-1" {
    #key_name = "machine_instance.${count.index}"

    #ami            = "ami-0a13d44dccf1f5cf6"

    #instance_type = "t2.micro"

    #count = 3

#}
```

- Alternatively,

```
variable "elb_names" {

  type = list

  default = ["dev-loadbalancer",
"stage-loadbalanacer","prod-loadbalancer"]

}



resource "aws_iam_user" "lb" {

  name = var.elb_names[count.index]

  count = 3

  path = "/system/"

}
```

Find the Issue - Use Case:

- For some exam questions, you have to find what part of a terraform configuration file should be removed as good terraform practice, for example here:

```
Terraform {

        Backend “s3” {

                Bucket = “mybucket”
                Key      =  “path/to/key”

                Region = “us-east-1”

                Access_key = 1234

                Secret_key = 1234567890

        }

}
```

- From the above piece of code, it can be easily deduced that the access_key and secret_key are not required as the path to the key is already specified.

Terraform Lock

- If supported by the backend, terraform will lock the state file for all operations that affect it
- There is a force-unlock command to manually unlock the state fille if unlocking fails
    - Terraform force-unlock

Use Case: Resources Deleted Out of Terraform:

- This is more of a scenario-based question.
- “You have created an EC2 instance. Someone has modified it manually, what happens when terraform plan is ran?”
- Possible scenarios: The instance type is changed, the resource is terminated
    - Scenario 1: Terraform will attempt to revert the instance type to that of the desired state
    - Scenario 2: Terraform will look to create the resource once again

Resource Block:

- Resource blocks describe one or more infrastructure objects, such as virtual networks, compute instances etc.
- Each resource block declares a resource of a given type e.g. “aws_instance” along with a local identification name e.g. “web”.

Sentinel

- An embedded policy-as-code framework integrated with the HashiCorp Enterprise Products
- Used for various cases including:
    - Verification for tags on an instance
    - Verification of encryption methods being in place
- In general for Terraform Enterprise, the workload follows plan -> sentinel checks -> apply

Sensitive Data in State File

- When managing any sensitive data in Terraform, it's good practice to treat the state file as a whole as sensitive data.
- Terraform Cloud always encrypts the state at rest and protects it with TLS in transit
- Terraform Cloud is capable of tracking the identity of the user requesting the state file and logging any changes made to it
- If using a backend e.g. S3, encryption at rest is supported when encryption is active.

Dealing with Credentials in Config

- In general, hard coding credentials isn't good practice, as it poses large security risks.
- Credentials can be stored outside of the terraform configuration
- Storing credentials as environment variables is generally considered best practice as they aren't committed.

Remote Backend - Terraform Cloud

- Stores Terraform state and may be used to run operations in Terraform Cloud
- When using fill remote operations, tasks such as terraform apply can be executed in Terraform Cloud's run environment; log outputs viewed on local machine

Misc:

- Terraform doesn't require go as a prerequisite
- Terraform works well with Linux, Windows and MAC

- Windows Server isn’t required for usage

Terraform Graph

- A command used to generate a visual representation of either a configuration or execution format
- The output format of terraform graph is the DOT format, which can “easily” be converted to another format for visualization

Splat Expressions

- Allows quick access to a list of all attributes
- Will output all possible values within the configuration where applicable
- Referencing Modules:

    ```
    resource "aws_instance" "example" {
      ami           = "ami-abc123"
      instance_type = "t2.micro"

      ebs_block_device {
        device_name = "sda2"
        volume_size = 16
      }
      ebs_block_device {
        device_name = "sda3"
        volume_size = 20
      }
    }
    ```

- The arguments of the ebs_block_device nested blocks can be accessed using a splat expression. For example, to obtain a list of all of the device_name values, use ***aws_instance.example.ebs_block_device[*].device_name.***
- The nested blocks in this particular resource type do not have any exported attributes, but if ebs_block_device were to have a documented id attribute then a list

of them could be accessed similarly as

aws_instance.example.ebs_block_device[*].id.

Terraform Technologies

- In any given terraform resource block, there are four main term types used:
- Resource Type: Specifies what resource is being defined in the resource block
- Example: Local identification name for the resource
- Ami, access_key etc.: Argument name/attribute
- Abc1234: Argument/Attribute value

Provider Configuration

- A block for provider configuration isn't mandatory for all terraform configurations
- When no resources are to be created, no provider needs to be specified

Terraform Output

- Used to extract the value of an output variable from a state file e.g. terraform output argument_name

Terraform Unlock

- If supported by your backend, terraform will lock the state file for all operations that could write to the state file
- Not all backends have locking functionality
- Terraform has a force-unlock command if unlocking failed
- ***Terraform force-unlock LOCK_ID [DIR]***

Misc Pointers - 01

- There are 3 primary benefits of Infrastructure as Code tools:
  - Automation
  - Versioning
  - Reusability
- There are various IaC tools available, such as:

    - Terraform
    - CloudFormation
    - Azure Resource Manager
    - Google Cloud Deployment Manager

Misc Pointers - 02

- Sentinel Is a proactive service
- Terraform doesn't modify the infrastructure, only the state file
- Slice function is not a function to be used with strings, unlike join, split, chomp
- It's not necessary to include the module version when pulling code from Terraform registry

Misc Pointers - 03

- Overuse of dynamic blocks can make configurations unreadable
- Terraform apply can change, destroy and provision resources, but not import any resources.

Terraform Enterprise and Cloud

- Terraform Enterprise allows several additional features not included in cloud, such as:
    - Single Sign-On
    - Auditing
    - Private Data Center
    - Clustering
- It should also be noted that Team and Governance features are not available for free on Terraform Cloud

Variables with Undefined Values

- If a variable is included with an undefined value, an error will not immediately occur
- Terraform will ask for you to supply a value to be associate with the undefined variable

Environment Variables

- Can be used to set variables
- Environment variables must be of format: TV_VAR_name e.g.
    - Export TF_VAR_region=us-west-1
    - Export TF_var_alist='[1,2,3]'

Structural Data Types

- A structural type allows multiple values of several distinct types to be grouped together as a single value
- List contains multiple values of the same type while object can contain multiple values of different types

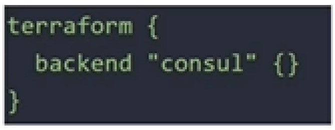
Structural Type	Description
Object	A collection of named attributes that each have their own type Object({<ATTR NAME> = <TYPE>, ... })
tuple	tuple ([<TYPE>, ...])

Backend Configuration

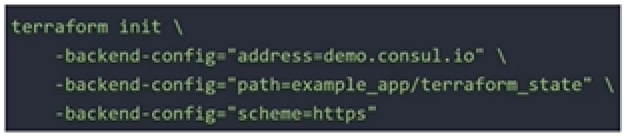
- Backends are configured directly in terraform files in the terraform block
- Once configured, they must be initialised

```
terraform {

  backend "s3" {

    bucket          = "demoremotebackend"

    key             = "remotedemo.tfstate"

    region          = "eu-west-2"

    access_key      = "AKIAIDHLXIC5PHKA4HRA"

    secret_key      = "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"

    #dynamodb_table = "s3-state-lock"
  }

}
```

Backend Configuration Types

- First-Time Configuration
  - When configuring a backend for the first time (moving from no defined backend to an explicitly configured one), terraform will give an option to migrate the currently-existing state to the new backend.
  - Allows adoption of backends without losing data in the state file
- Partial Configuration
  - Not every required argument needs to be specified for a backend configuration
  - It may be better to omit certain arguments to avoid storing secrets within the main configuration
  - With a partial configuration, the remaining arguments must be provided as part of the initialization process
  - 

```
terraform {
  backend "consul" {}
}
```



```
terraform init \
    -backend-config="address=demo.consul.io" \
    -backend-config="path=example_app/terraform_state" \
    -backend-config="scheme=https"
```

Terraform Taint

- Manually marks a Terraform-managed resource as tainted, forcing it to be destroyed and recreated on the next apply
- Once a resource is marked as tainted, the next plan will show the resource will be destroyed and recreated; the next apply run implements this change

Provisioner - Local

- The local-exec provisioner invokes commands to be ran on your local machine after a resource is created
- Defined in a similar manner to:

```
resource "aws_instance" "myec2" {
  ami           = "ami-0a13d44dccf1f5cf6"

  instance_type = "t2.micro"


  provisioner "local-exec" {

    command = "echo ${aws_instance.myec2.private_ip} >> privateips.txt"

  }

}
```

Provisioner - Remote

- Remote-exec provisioner invokes a script on a remote resource after it’s created
- Supports multiple connection types including ssh and winrm
- Invoked via configurations similar to:

```
  provisioner "remote-exec" {

    inline = [

      "sudo amazon-linux-extras install -y nginx1.12", #install nginx

      "sudo systemctl start nginx"                     # start nginx

    ]

    connection {

      #connection method

      type = "ssh"

      user = "ec2-user"

      #private key for authentication

      private_key = file("./remote-exec-keypair.pem")

      host        = self.public_ip

    }
  }

}
```

Provisioner - Failure Behaviour

- By default provisioners that fail will cause the Terraform apply to fail
- This behaviour can be altered by changing the “on_failure” setting to one of 2 values:
    - Continue - ignore the error and continue with the resource creation
    - Fail - raise an error and stop the apply (default behaviour)
        - If this is a creation provisioner, taint the resource

Provisioner Types

- There are two types of provisioners:
    - Creation-TIme
        - Only run during creation, not during updating or any other lifecycle
        - If a creation-time provisioner fails, the resource is marked as tainted
    - Destroy-Time
        - Ran before the resource is destroyed

Input Variables

- The value associated with a variable can be assigned via multiple methods:
- Value associated can be defined either by the CLI or the tfvars vile
- To load a custom tfvars file: terraform apply -var-file=”filename.tfvars”, this doesn’t need to be applied if terraform.tfvars is the only variable file

Variable Definition Precedence

- Terraform loads variables in the following order, with the importance increasing as the list progresses:

1. Environment variables
2. Terraform.tfvars if present
3. Terraform.tfvars.json if present
4. Any *.auto.tfvars or *.auto.tfvars.json files processed alphabetically
5. Any -var or -var-file options on the command line in the order they’re provided

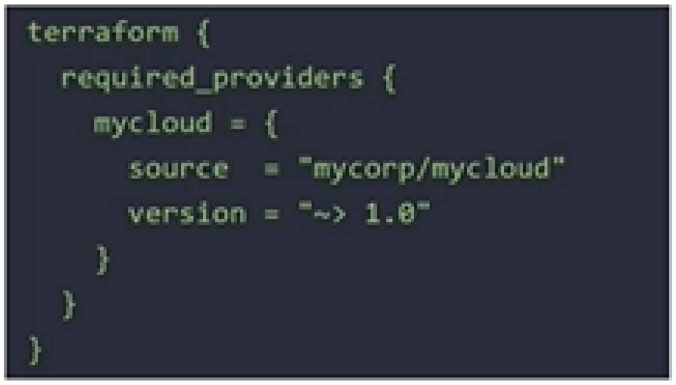
- If a variable is defined multiple times with different values, the last definition will be the applied value

Local Backend

- Stores the state on the local filesystem
- Locks the state using system APIs
- Performs operations locally
- The default backend used by terraform
- Path should be specified where appropriate

Required Providers

- Each terraform module must declare which providers it requires so Terraform can install and use them
- Provider requirements are declared in a required_providers block in a similar manner to:



```
terraform {
  required_providers {
    mycloud = {
      source  = "mycorp/mycloud"
      version = "~> 1.0"
    }
  }
}
```

Required Version

- The required_version setting accepts a version constraint string, specifying the version of terraform to be used with your configuration
- If the running of terraform doesn't match the specified constraints, terraform will produce an error and exit without applying any more changes

Version Arguments

Version Argument	Description
>= x.y	Greater than or equal to x.y
<	Less than or equal to x.y
~>x.y	Any version in the range x.y
>=X.Y,<=A.B	Any version in the range X.Y - A.B

Fetching Values from Map

- To reference a particular value from a map: var.<variable_id>[“key”]

Terraform and GIT

- In practice, careful consideration should be taken when committing terraform code. The .gitignore should be configured to ignore files which may contain sensitive information, such as:
  - Terraform.tfstate
  - *.tfvars
- Arbirtrary Git repositories can be used by prefixing the address with a special git:: prefix
  - Any valid git URL can be specified to select one of the protocols supported by GIT
- By default, Terraform will clone and use the default branch (HEAD) in the selected repository
  - This can be overwritten by adding the ref argument e.g.:
    - Module “vpc” {
      - Source = “git::/https://example.com/repo.git?ref=v1.0”
    - }
  - The value of the ref argument can be any reference that would be accepted by the git checkout command, including branch and tag names

Terraform Workspace

- Command used to manage workspaces within terraform

- Each workspace has its own state file directory
- Commonly used for separating environments such as stage or prod
    - Doesn't provide a strong separation between environments
- To create a new workspace: terraform workspace new <workspace_name>
- To switch to a new workspace: terraform workspace select <workspace_name>
    - Automatically switch to workspace when new one is created

Dependency Types - Implicit

- With implicit dependency, terraform can automatically find references of the object and create an implicit ordering requirement between the two resources
- Example: for creating an eip to be associated with an EC2 instance, for the public_ip in the EC2 resource block, can specify: aws_eip_myeip.private_ip
    - Creates eip first to apply to ec2 instance

Dependency Types - Explicit

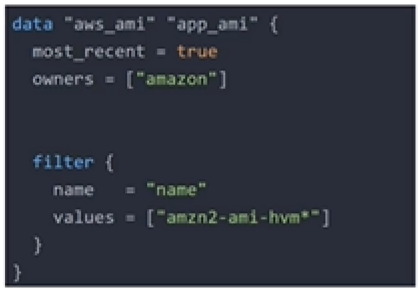
- Explicitly specifying a dependency is only required when a resource relies on another's behaviour but doesn't access any of that resource's data
- Can add depends_on = [<resource_type>.<resource_id>] to add explicit dependencies

State Command

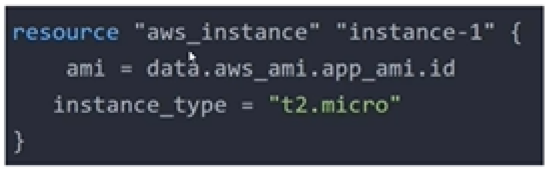
- Terraform state list
    - List resources within terraform state
- Terraform state mv
    - Move items within terraform state
    - Can be used for renaming
- Terraform state pull
    - Manually download and output the state from state file
- Terraform state rm
    - Remove items from the state file
- Terraform state show
    - Show the attributes of a single resource in the Terraform state

Data Source Code

- Data sources allow data to be fetched or computed for use elsewhere within the configuration
- Reads from a specific data source and exports the results under a particular value:



```
data "aws_ami" "app_ami" {
  most_recent = true
  owners = ["amazon"]


  filter {
    name    = "name"
    values = ["amzn2-ami-hvm*"]
  }
}
```



```
resource "aws_instance" "instance-1" {
    ami = data.aws_ami.app_ami.id
   instance_type = "t2.micro"
}
```

Terraform Taint

- Can be used to taint resources within a module
- Terraform taint [options] address
- For multiple sub modules, apply the following syntax:
- Module.foo.module.bar.resource_type.resource_id

Terraform Plan Destroy

- The behaviour of any terraform destroy command can be previewed with terraform plan -destroy

Terraform Module Sources

- The module installer supports installation from a number of different source types, such as local paths, the terraform registry etc.
- Local path references allow for factoring out portions of configuration within a single source repository
- A local path must begin with either ./ or ../ etc to indicate a local path is intended

Larger Infrastructure

- Cloud providers have certain amounts of rate limiting; meaning only a certain number of resources can be requested over a period of time

- It’s recommended that larger configurations are broken into multiple smaller sets that can be independently applied
- Alternatively, can use the -refresh=false and target flags, though this isn’t recommended.

Misc Pointers

- Lookup retrieves the value of a single element from a map
    - lookup(map, key, default)
- Various commands run terraform refresh implicitly, such as terraform [plan, apply, destroy]
- Terraform init and import don’t refresh the state implicitly
- Array data type isn’t supported
- Various variable definition files can automatically be loaded such as
    - Terraform.tfvars
    - terraform.tfvars .json
    - Any files with .auto.tfvars.json
- Both implicit and explicit dependencies are stored in terraform.tfstate
- Terraform init -upgrade automatically upgrades all previously installed plugins to their newest versions
- The terraform console command provides an interactive console for evaluating expressions
- The declaration of variables differs between terraform 0.11 and 0.12:
    - 0.11: “${var.instance_type}”
    - 0.12: var.instance_type