TERRAFORM

Section-1: Introduction to Terraform:

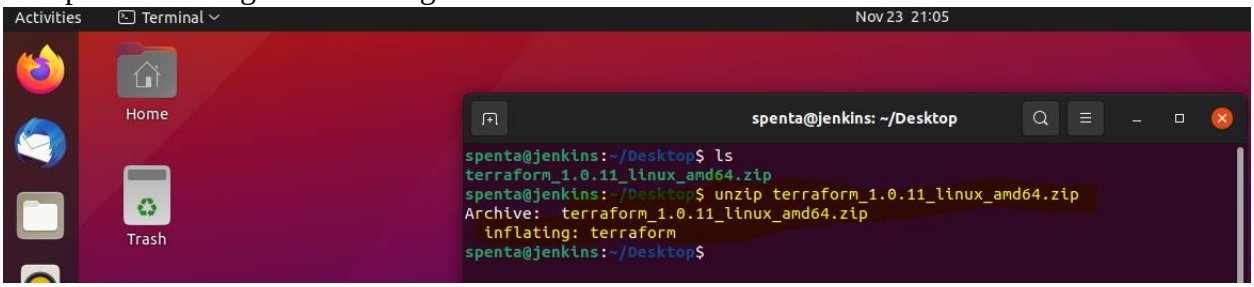
1. **Terraform** is an infrastructure as code (IaC) tool that allows you to build, change, and version infrastructure safely and efficiently.
    - Terraform is Infrastructure as Code.
    - Automation of your infrastructure.
    - Keep your Infrastructure in certain state (Compliant)
    - Make your infrastructure Auditable.
        - ➢ You can keep your infrastructure change history in a version control system like GIT.
    - Ansible, Chef, Puppet, Saltstack have a focus on automating the installation and configuration of software.
        - ➢ Keeping the machines in compliance, in a certain state.
    - Terraform can automate provisioning of the infrastructure itself.
        - ➢ Works well with automation software like ansible to install software after the infrastructure is provisioned.

1.2: Terraform Installation on Linux:

1. Download the Terraform from the terraform site based on the Operation System.

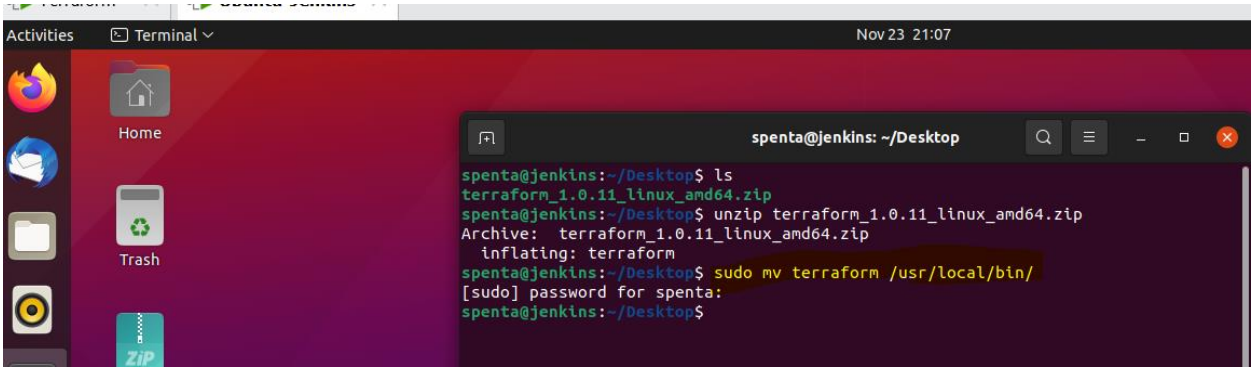
   https://www.terraform.io/downloads.html

2. Copy the Zip file on the server where we need to install.
3. Unzip the file using the following command.

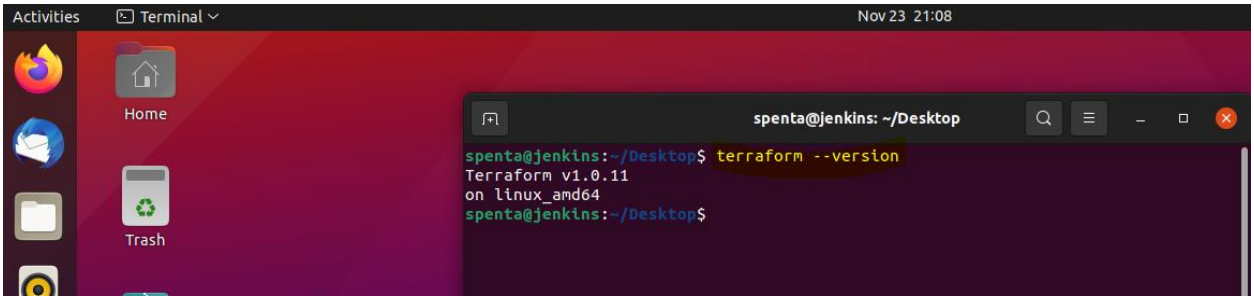


4. Move the executable into a directory searched for executables.

   sudo mv terraform /usr/local/bin/



5. Verify the Terraform using the following command.
   Terraform –version.

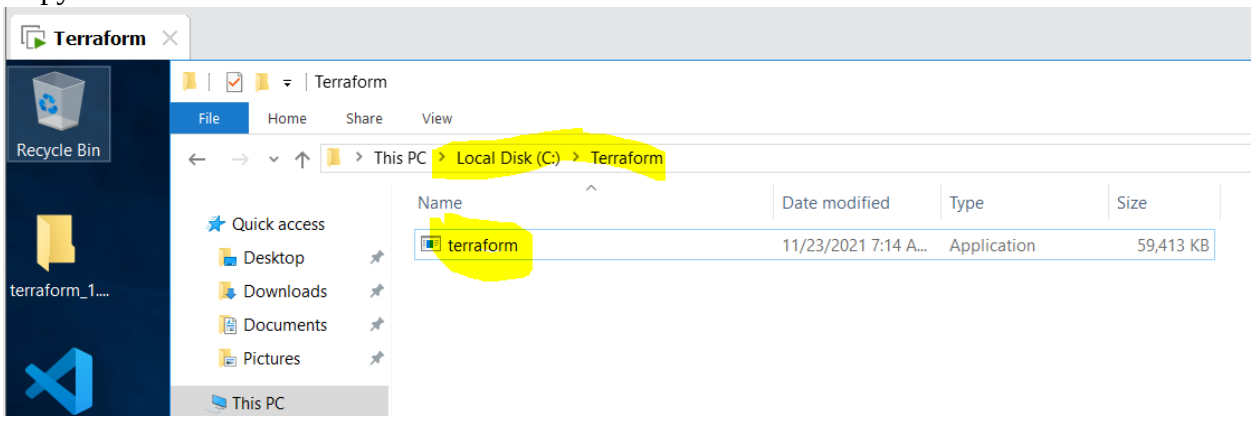


1.3: Terraform Installation on Windows:

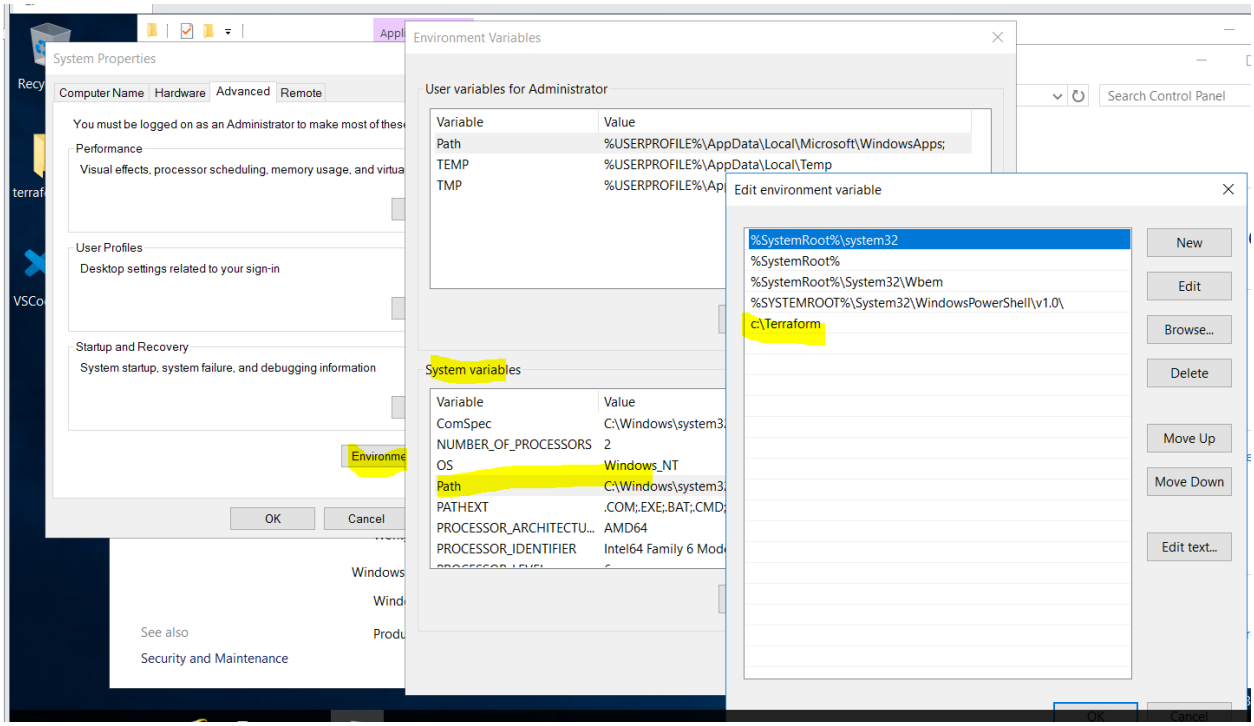
1. Download the Terraform from the terraform site based on the Operation System.

   https://www.terraform.io/downloads.html

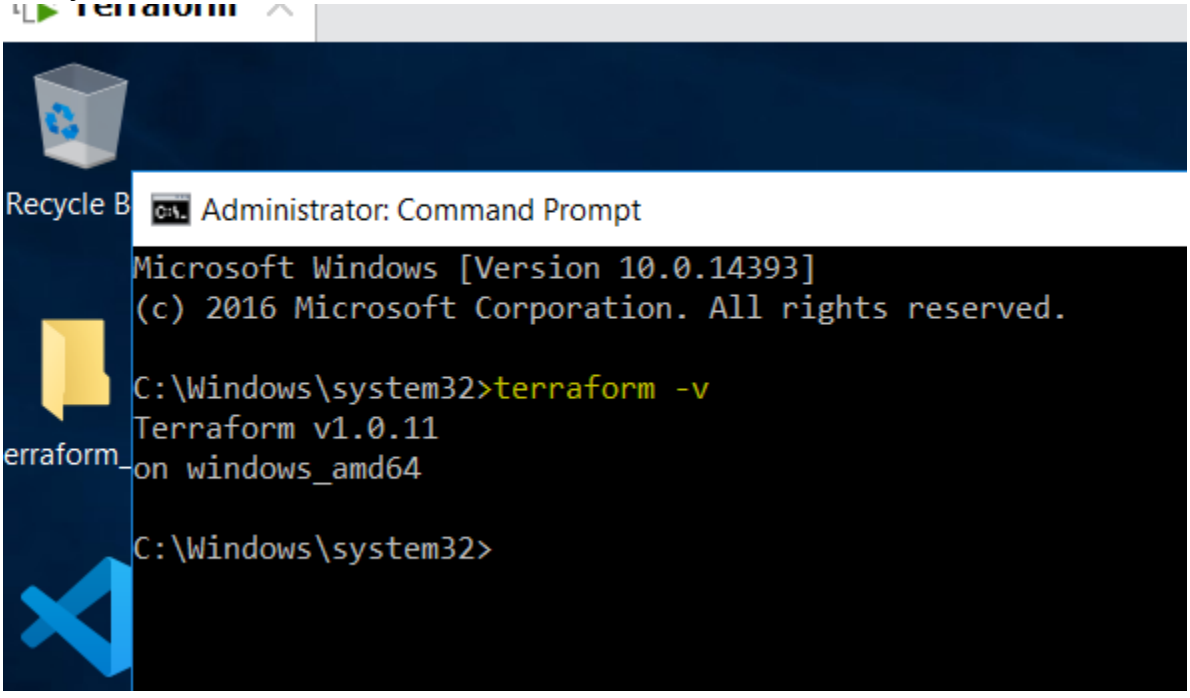
2. Copy the file from Downloads to C:\Terraform folder.



3. Create Environment variable as shown below.



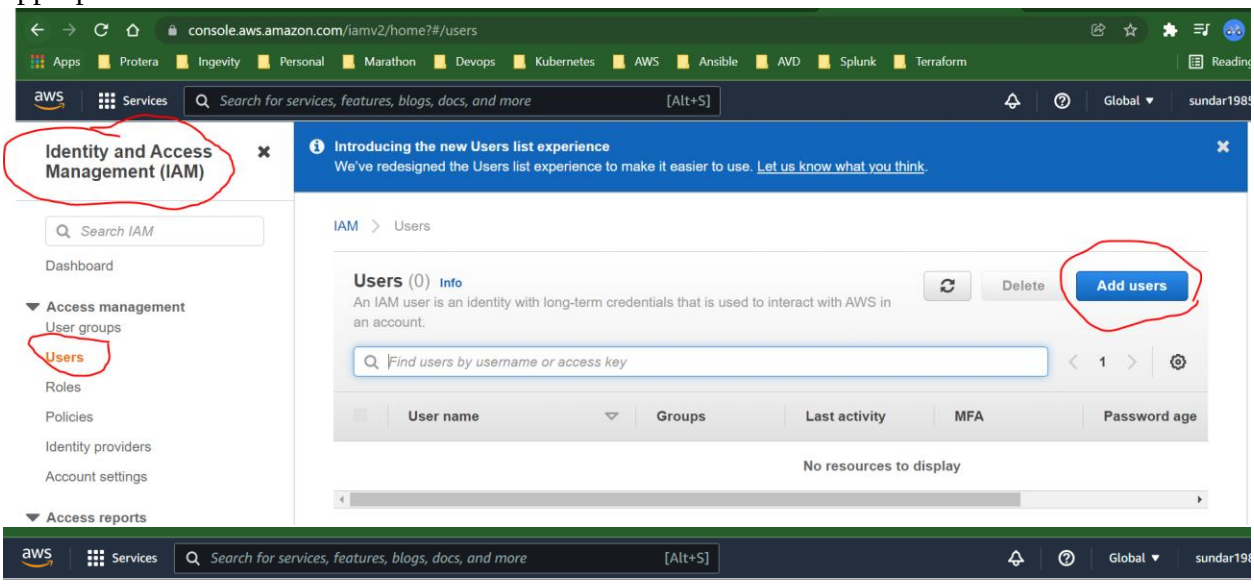
4. Verify the terraform

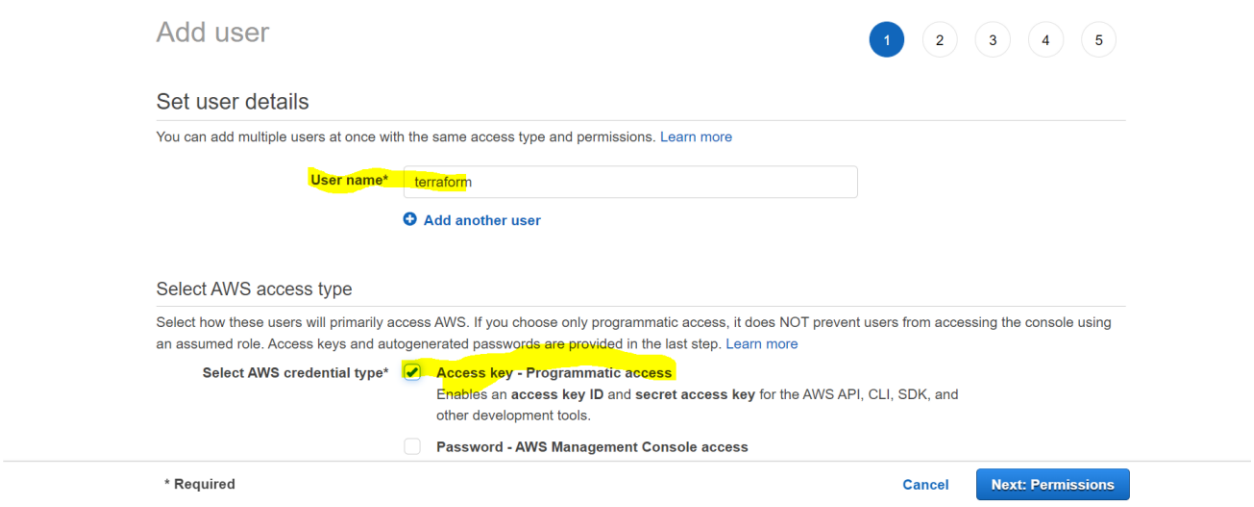


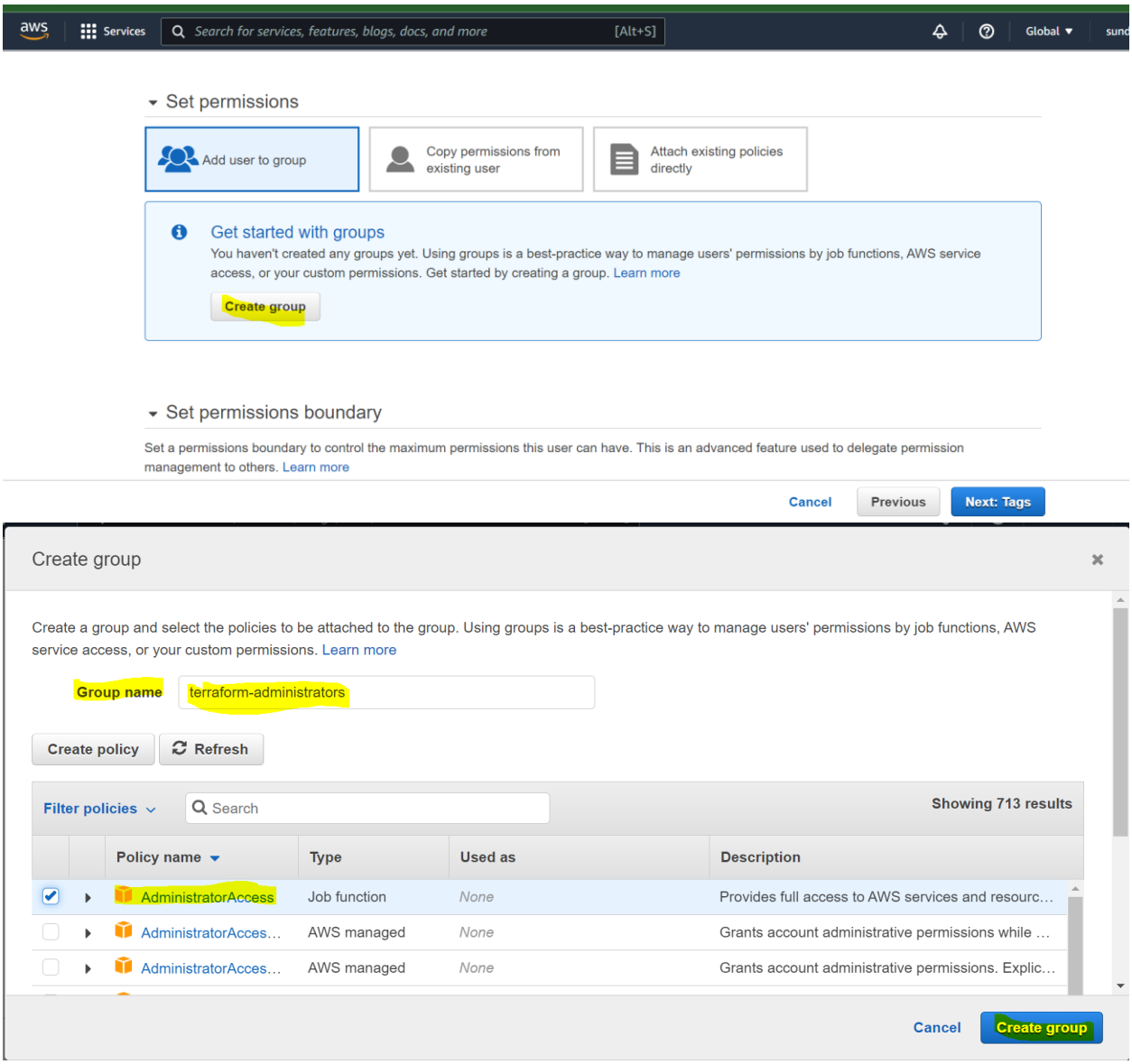
Section-2: Terraform Basics:

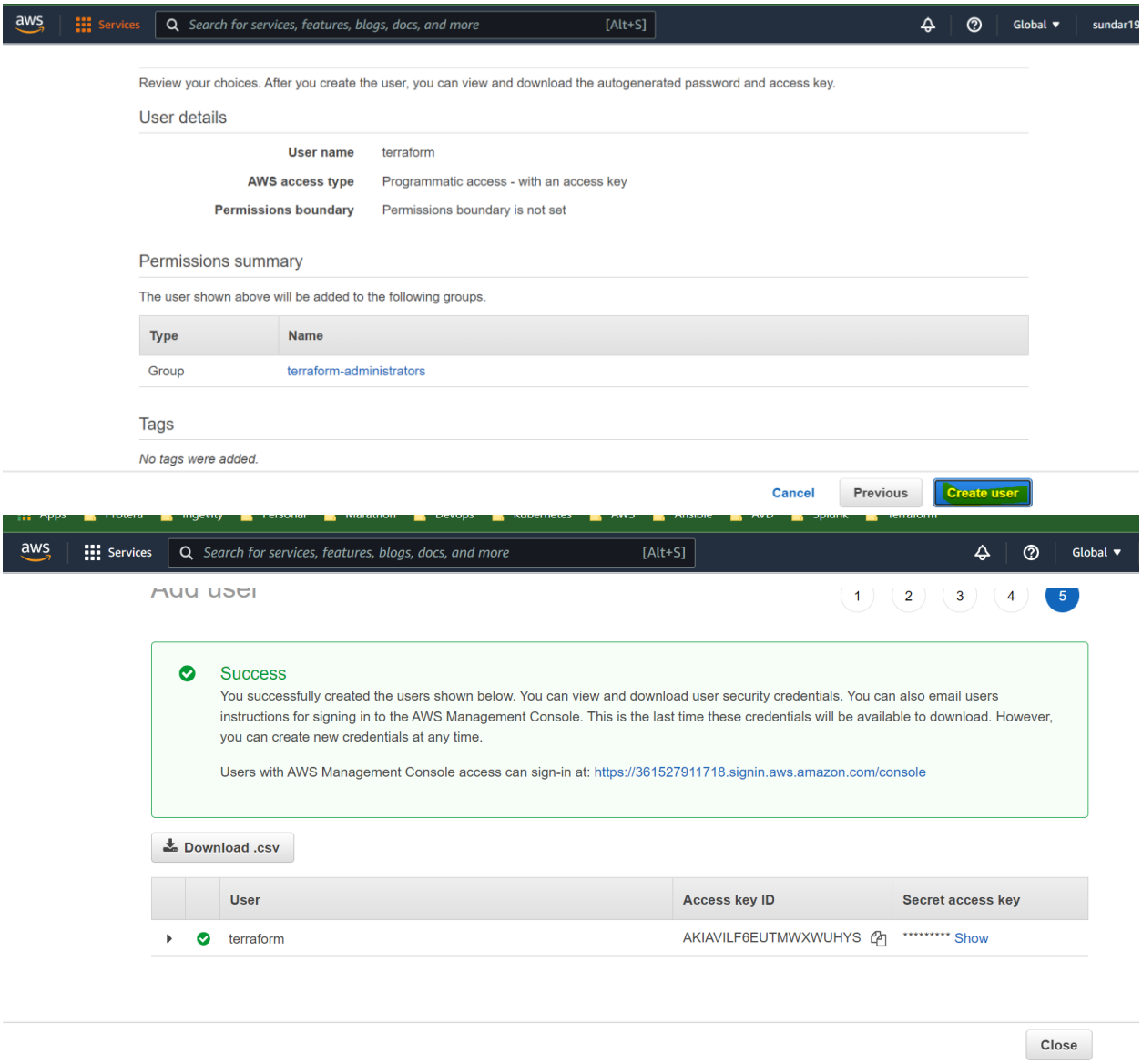
2.1: Understanding the Terraform HCL (Hashicorp Configuration Language):

1. Spinning up an instance on AWS
    - Open AWS account
    - Create an IAM Admin User.
    - Create a terraform file to spin up t2.micro instance
    - Run Terraform apply.
2. Please login to the AWS Management console using your id and password.
3. Go to the service called IAM → Create a new user called “terraform” → Provide the appropriate access to terraform user.






aws
Services
Search for services, features, blogs, docs, and more
[Alt+S]
Global
Set permissions
Add user to group
Copy permissions from existing user
Attach existing policies directly
Get started with groups
You haven't created any groups yet. Using groups is a best-practice way to manage users' permissions by job functions, AWS service access, or your custom permissions. Get started by creating a group. Learn more
Create group
Set permissions boundary
Set a permissions boundary to control the maximum permissions this user can have. This is an advanced feature used to delegate permission management to others. Learn more
Cancel
Previous
Next: Tags
Create group
Create a group and select the policies to be attached to the group. Using groups is a best-practice way to manage users' permissions by job functions, AWS service access, or your custom permissions. Learn more
Group name
terraform-administrators
Create policy
Refresh
Filter policies
Search
Showing 713 results
Policy name
Type
Used as
Description
AdministratorAccess
Job function
None
Provides full access to AWS services and resourc…
AdministratorAcces…
AWS managed
None
Grants account administrative permissions while …
AdministratorAcces…
AWS managed
None
Grants account administrative permissions. Explic…
Cancel
Create group



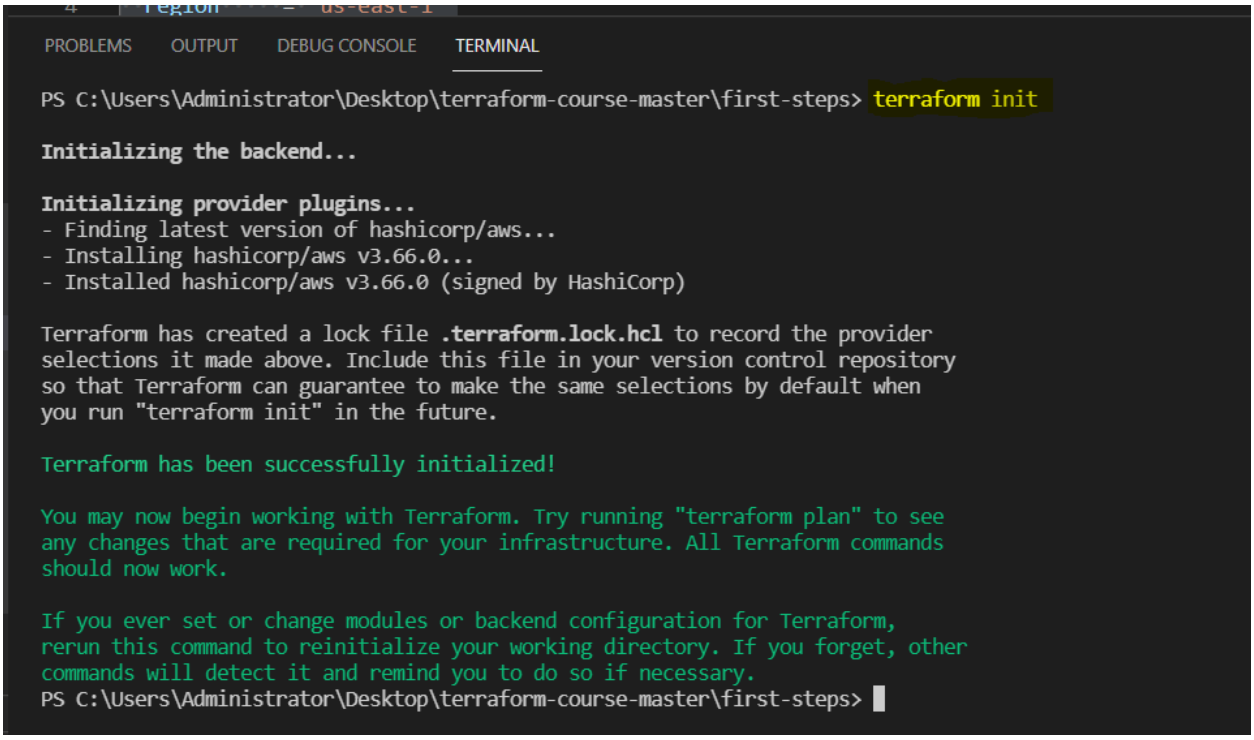
5. Please download the access key file and save it.

2.2: Spinning up the instance using terraform in AWS:

1. Install the MS Visual Studio in Windows machine and install the Terraform plugin in Visual Studio code.
2. Create file name called instance.tf in VS code as shown below.

```
provider "aws" {
  access_key = "Your Key"
  secret_key = "Your secret key"
  region     = "us-east-1"
}

resource "aws_instance" "example" {
  ami           = "ami-0279c3b3186e54acd"
  instance_type = "t2.micro"
}
```

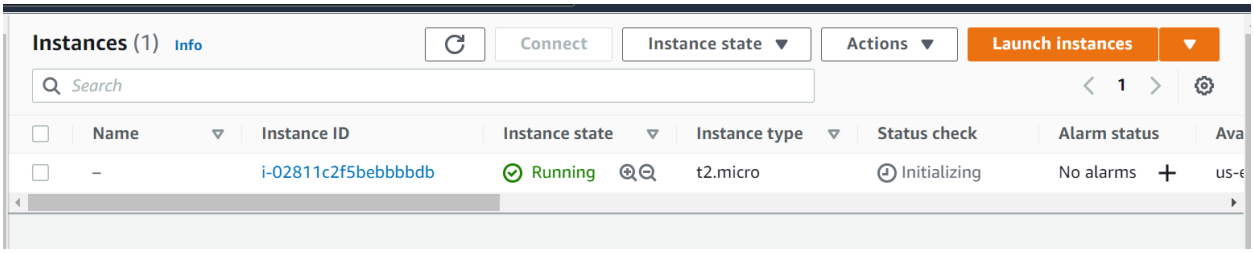
3. Initialize the terraform using the following command.
   /> terraform init.



4. Now run the terraform apply using the following command.
   /> terraform apply

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL

          + tags                  = (known after apply)
          + throughput            = (known after apply)
          + volume_id             = (known after apply)
          + volume_size           = (known after apply)
          + volume_type           = (known after apply)
        }
    }

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

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

  Enter a value: yes

aws_instance.example: Creating...
aws_instance.example: Still creating... [10s elapsed]
aws_instance.example: Still creating... [20s elapsed]
aws_instance.example: Still creating... [30s elapsed]
aws_instance.example: Still creating... [40s elapsed]
aws_instance.example: Still creating... [50s elapsed]
aws_instance.example: Creation complete after 59s [id=i-02811c2f5bebbbbdb]

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
PS C:\Users\Administrator\Desktop\terraform-course-master\first-steps>
```

5. We can validate whether the instance is created in AWS console.

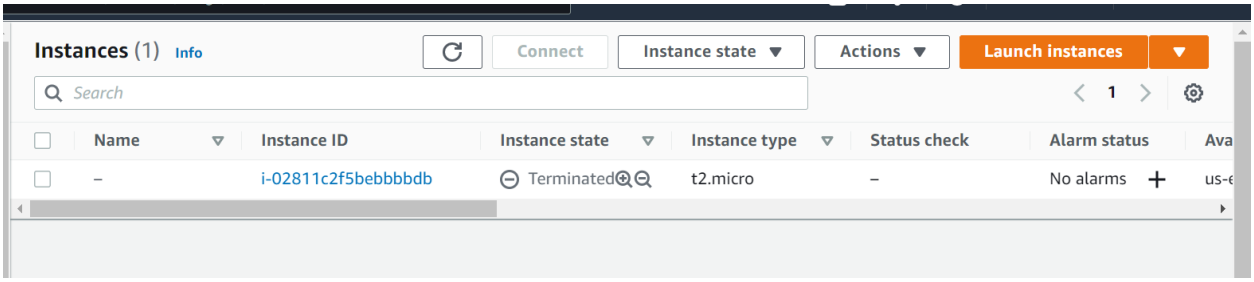


6. We can delete this instance using the below command.

   /> terraform destroy.

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL

          - iops                  = 100 -> null
          - tags                  = {} -> null
          - throughput            = 0 -> null
          - volume_id             = "vol-0f58643293c0939c7" -> null
          - volume_size           = 8 -> null
          - volume_type           = "gp2" -> null
        }
    }

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

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

  Enter a value: yes

aws_instance.example: Destroying... [id=i-02811c2f5bebbbbdb]
aws_instance.example: Still destroying... [id=i-02811c2f5bebbbbdb, 10s elapsed]
aws_instance.example: Still destroying... [id=i-02811c2f5bebbbbdb, 20s elapsed]
aws_instance.example: Still destroying... [id=i-02811c2f5bebbbbdb, 30s elapsed]
aws_instance.example: Still destroying... [id=i-02811c2f5bebbbbdb, 40s elapsed]
aws_instance.example: Destruction complete after 44s

Destroy complete! Resources: 1 destroyed.
PS C:\Users\Administrator\Desktop\terraform-course-master\first-steps>
```

7. Validate the same in AWS console.



8. Use the different commands in terraform.
   /> terraform plan -out out.terraform
   /> terraform apply "out.terraform"
9. Testing what infrastructure would be build based on our terraform files.
   /> terraform init.
10. Then use terraform apply to execute the changes.
    />terraform apply
11. Keeping the changes in an out file.
    />terraform plan -out changes.terraform
12. Then only apply those changes to the infrastructure changes.
    />terraform apply changes.terraform
    />rm changes.terraform
13. Afterwards, we can destroy the infrastructure.
    />terraform destroy

2.3: Terraform variable Types:

1. Terraform variables were completely reworked for the terraform 0.12 release.
2. You can now have more control over the variables, and have for and for-each loops, which where not possible with earlier versions.
3. You don't have to specify the type in variables, but it's recommended.
4. Terraform's simple variable types.
    - Number
    - String
    - Bool

```
variable "a-string"{
  type = string
}
variable "this-is-a-number"{
  type = number
}
variable "true-or-false"{
  type = bool
}
```

5. Terraform's complex types
    - List(type)
    - set(type)
    - Map(type)
    - Object({<ATTR NAME> = <TYPE>,..})
    - Tuple([<TYPE>,...])
6. List & Map we already covered during a previous demo.
    - List:[0,1,5,2]
    - Map: {"key" = "value"}
7. A list is always ordered, it will always return 0,1,5,2 and not 5,1,2,0.
8. A "set" is like a list, but it doesn't keep the order you put it in, and can only contain unique values.
    - A list that has [5,1,1,2] becomes [1,2,5] in a set.
9. An object is like a map, but each element can have a different type.
10. A tuple is like a list, but each element can have a different type.
11. The most common types are list and map, the other ones are only used sporadically.
12. The ones you should remember are the simple variable types string, number, bool and the list & map.
13. You can also let terraform decide on the type.

2.4: Variables in Terraform:

providers.tf

```
provider "aws"{
  access_key = "${var.AWS_ACCESS_KEY}"
  secret_key = "${var.AWS_SECRET_KEY}"
  region = "${var.AWS_REGION}"
}
```

vars.tf

```
variable "AWS_ACCESS_KEY"{}
variable "AWS_SECRET_KEY"{}
variable "AWS_REGION"{
  default = "eu-west-1"
}
```

terraform.tfvars

```
AWS_ACCESS_KEY = ""
AWS_SECRET_KEY = ""
AWS_REGION = ""
```

instance.tf

```
resource "aws_instance" "web"{
```

```
  ami = "ami-0d729a60"
  instance_type = "t2.micro"
}
```

vars.tf

```
variable "AWS_ACCESS_KEY"{}
variable "AWS_SECRET_KEY"{}
variable "AWS_REGION"{
  default = "eu-west-1"
}
variable "AMIS"
  type = "map"
  default = {
  us-east-1 = "ami-13be557e"
  us-west-2 = "ami-06b94666"
  us-west-1 = "ami-0d729a60"
 }
}
```

instance.tf

```
resource "aws_instance" "web"{
  ami = "${lookup(var.AMIS, var.AWS_REGION)}"
  instance_type = "t2.micro"
}
```

https://cloud-images.ubuntu.com/locator/ec2/

2.5: Software Provisioning:

There are 2 ways to provision software on your instances.

1. You can build your own custom AMI and bundle your software with the image.
    - Packer is a great toll to do this.
2. Another way is to boot standardized AMIs and then install the software on it you need.
    - Using file uploads
    - Using remote exec
    - Using automation tools like Chef, Puppet, Ansible
3. Current state for terraform with automation (Q4 2016):
    - Chef is integrated with terraform, you can add chef statements.
    - You can run Puppet agent using remote-exec
    - For Ansible, you can first run terraform and output the IP addresses then run Ansible-Playbooks on those hosts.
        - ➢ This can be automated in a workflow script.
        - ➢ There are 3rd party initiatives integrating Ansible with terraform

File Uploads:

```
resource "aws_instance" "web" {

        ami = "${lookup(var.AMIS, var.AWS_REGION)}"

       instance_type = "t2.micro"


provisioner "file"{

        source = "app.conf"

        destination = "/etc/myapp.conf"

}

}
```

1. File uploads is an easy way to upload a file or a script.
2. Can be used in conjunction with remote-exec to execute a script.
3. The provisioner may use SSH (Linux hosts) or WinRM (On windows hosts).
4. To override the SSH defaults, you can use "connection"

```
resource "aws_instance" "web" {
```

```
        ami = "${lookup(var.AMIS, var.AWS_REGION)}"
        instance_type = "t2.micro"

    provisioner "file"{
        source = "script.sh"
        destination = "/opt/script.sh"
        connection{
            user = "${var.instance_username}"
            password = "${var.instance_password}"
                }
          }
    }
```

5. When spinning up instances on AWS, ec2-user is the default user for Amazon Linux and Ubuntu.
6. Typically, on AWS, you will use SSH keypairs:

```
        resource äws_key_pair""edward-key"{
            key_name = "mykey"
            public_key = "ssh-rsa my-public-key"
        }

        resource "aws_instance" "web" {
             ami = "${lookup(var.AMIS, var.AWS_REGION)}"
            instance_type = "t2.micro"
            key_name = "${aws_key_pair.mykey.key_name}"
        provisioner "file"{
            source = "script.sh"
             destination = "/opt/script.sh"
             connection{
                    user = "${var.instance_username}"
                    private_key = "${file(${var.path_to_private_key})}"
          }
         }
        }
```

7. After you uploaded a script, you will want to execute it.
8. You can execute a script using remote-exec:

```
        resource "aws_instance" "web" {
            ami = "${lookup(var.AMIS, var.AWS_REGION)}"
            instance_type = "t2.micro"

        provisioner "file"{
```

```
        source = "script.sh"
        destination = "/opt/script.sh"
    }

    provisioner "remote-exec"{

        inline = [
          "chmod +x /opt/script.sh",
          "/opt/script.sh arguments"
        ]
      }
    }
```

2.6: Outputting Attributes:

1. Terraform Keeps attributes of all the resources you create.
   E.g: The aws_instance resource has the attribute public_ip

2. Those attributes can be queried and outputted.
3. This can be useful just to output valuable information or to feed information to external software.
4. Use “output" to display the public IP address of an AWS resource:

```
resource äws_instance" "ëxample" {

    ami = "${lookup(var.AMIS, var.AWS_REGION)}"

    instance_type = "t2.micro"

}


output "ip" {

    value = "${aws_instance.example.public_ip}"

}
```

5. You can refer to any attribute by specifying the following elements in your variables:
   - The resource type: aws_instance
   - The resource name: example
   - The attribute name: public_ip

6. You can also use the attributes in a Script:

```
resource "aws_instance" "example" {
        ami = "${lookup(var.AMIS, var.AWS_REGION)}"
        instance_type = "t2.micro"
        provisioner "local-exec"{
                command = "echo
${aws_instance.example.private_ip}>>private_ips.txt"
 }
}
```

7. Useful for instance to start automation scripts after infrastructure provisioning.
8. You can populate the IP addresses in an ansible host file.
9. Or another possibility: execute a script (with attributes as argument) which will take care of a mapping of resource name and the IP address.

2.7: Remote State:

1. Terraform keeps the remote state of the infrastructure.
2. It stores it in a file called terraform.tfstate.
3. There is also a backup of the previous state in terraform.tfstate.backup.
4. When you execute terraform apply, a new terraform.tfstate and backup is written.
5. This is how terraform keeps track of the remote state.
    - if the remote state changes and you hit terraform apply again, terraform will make changes to meet the correct remote state again.
    - E.g. you terminate an instance that is managed by terraform, after terraform apply it will be started again.
6. You can keep the terraform.tfstate in version control.
   E.g GIT
7. It gives you a history of your terraform.tfstate file
8. It allows you to collaborate with other team members.
    - Unfortunately, you can get conflicts when 2 people work at the same time.
9. Local state works well in the beginning, but when you project becomes bigger, you might want to store your state remote.
10. The terraform state can be saved remote, using the backend functionality in terraform.
11. The default is a local backend (the local terraform state file)

12. Other backends include:

    - S3(with a locking mechanism using DynamoDB)
    - consul (with locking)
    - terraform enterprise (the commercial solution)

13. Using the backend functionality has benefits:
    - Working in a team: it allows for collaboration; the remote state will always be available for the whole team.
    - The state file is not stored locally. Possible sensitive information is now only stored in the remote state.
    - Some backends will enable remote operations. The terraform apply will the run completely remote. These are called the enhanced backends. (https://www.terraform.io/docs/backends/types/index.html)

14. There are 2 steps to configure a remote state:
    - Add the backend code to a .tf file.
    - Run the initialization process

15. To configure a consul remote store, you can add a file backend.tf with the following contents:

```
Terraform {
        backend "consul"{
          address = "demo.consul.io" #hostname of consul cluster
          path ="terraform/myproject"
        }
}
```

16. You can also store your state in S3:

```
terraform {
        backend "s3"{
          bucket = "mybucket"
          key = "terraform/myproject"
          region = "eu-west-1"
        }

}
```

17. When using an S3 remote state, it's best to configure the AWS credentials:

```
$aws configure
AWS Access Key ID[]: AWS-key
AWS Secret Access Key[]: AWS_secret_key
Default region name[]: eu-west-1
Default output format[None]
```

18. Next step, terraform init
19. Using a remote store for the terraform state will ensure that you always have the latest version of the state.
20. It avoids having to commit and push the terraform.tfstate to version control.
21. Terraform remote stores don't always support locking
    - The documentation always mentions if locking is available for a remote store.
    - S3 and consul support it.
22. You can also specify a (read-only) remote store directly in the .tf file.

```
data "terraform_remote_state" "aws_state" {
        backend = "s3"
        config {
          bucket = "terraform-state"
          key = "terraform.tfstate"
          access_key = "${var.AWS_ACCESS_KEY}"
          secret_key = "${var.AWS_SECRET_KEY}"
          region = "${var.AWS_REGION}"
        }
}
```

23. This is only useful as a read only feed from your remote file.
24. It's a data source.
25. It is very useful to generate outputs.

2.8: Data Sources:

1. For certain providers (like AWS), terraform provides datasources.
2. Datasources provide you with dynamic information.
    - A lot of data is available by AWS in a structured format using their API.
    - Terraform also exposes this information using data sources.
    - EX: List of AMIs , List of availability Zones.
3. Another great example is the datasource that gives you all IP addresses in use by AWS.
4. This is great if you want to filter traffic based on an AWS region
    - E.g. Allow all traffic from amazon instances in Europe.
5. Filtering traffic in AWS can be done using security groups.
    - Incoming and outgoing traffic can be filtered by protocol, IP range, and port.
    - Similar to Iptables (Linux) or a firewall appliance.

```
data "aws_ip_ranges" "european_ec2" {
  regions  = ["eu-west-1", "eu-central-1"]
  services = ["ec2"]
}

resource "aws_security_group" "from_europe" {
  name = "from_europe"

  ingress {
    from_port   = "443"
    to_port     = "443"
    protocol    = "tcp"
    cidr_blocks = slice(data.aws_ip_ranges.european_ec2.cidr_blocks, 0, 50)
  }
  tags = {
    CreateDate = data.aws_ip_ranges.european_ec2.create_date
    SyncToken  = data.aws_ip_ranges.european_ec2.sync_token
  }
}
```

2.9: Templates:

1. The template provider can help creating customized configuration files.
2. You can build templates based on variables from terraform resource attributes (e.g. a public IP address)
3. The result is a string that can be used as a variable in terraform.
    - The string contains a template.
    - e.g. a configuration file
4. Can be used to create generic templates or cloud init configs.
5. In AWS, you can pass commands that need to be executed when the instance starts for the first time.
6. In AWS this is called "user-data"

7. If you want to pass user-data that depends on other information in terraform (e.g IP address), you can use the provide template.
8. First you can create a template file:
   #!/bin/bash
   echo "databse-ip" = ${myip} >>/etc/myapp.config

9. Then you create a template_file resource that will read the template file and replace ${myip}with the IP address of an AWS instance created by terraform.

   ```
   data "template_file" "my-template" {
   `       template = "${file("templates/init.tpl")}"

           vars{
             myip = "${aws_instance.database1.private_ip}
           }
   }
   ```
10. Then you can use the my-template resource when creating a new instance
    ```
    # Create a web server
    resource "aws_instance" "web" {
            # ---
            user_data = "${data.template_file.my-template.rendered}"
    }
    ```
11. When terraform runs, it will see that it first needs to spin up the databse1 instance, then generate the template and only then spin up the web instance.
12. The web instance will have the template injected in the user_data and when it launches, the user-data will create a file /etc/myapp.config with the IP address of the database.
13. First you create a template file:
    #!/bin/bash
    echo "database-ip = ${myip}" >>/etc/myapp.config

2.10: Other Providers:

1. Terraform is a toll to create and manage infrastructure resources.
2. Terraform has many providers to choose from
3. AWS is the most popular one and will be the one we will discuss in this course.
4. Potentially any company that opens up an API, can be used as a terraform provider.
5. Some other examples of cloud providers supported by terraform are:
   - Google Cloud
   - Azure
   - Heroku

- Digital Ocean

6. And on-premises / private cloud:
    - VMWare vCloud/ vSphere / OpenStack
7. It is not only limited to cloud providers:
    - Datadog – monitoring
    - GitHub - version control
    - Mailgun - emailing (SMTP)
    - DNSSimple / DNSMadeEAsy / UltraDNA -DNS hosting
8. The full list can be found at https://www.terraform.io/docs/providers/index.html
9. The terraform resources for other providers are very similar.

2.11: Modules:

1. You can use modules to make your terraform more organized.
2. Use third party modules.
    - modules from GitHub
3. Reuse parts of your code
    - e.g. to set up network in AWS -the Virtual private network (VPC)
4. Use a module from git

```
module "module-example"{
        source = "github.com/sundara/terraform-module-example"
}
```

5. Use a module from a local folder

```
module "module-example" {
        source = "./module-example"
}
```

6. Pass arguments to the module

```
module "module-example" {
        source = "./module-example"
        region = "us-west-1"
        ip-range = "10.0.0.0/8"
        cluster-size = "3"
}
```

7. Inside the module folder, you just have again terraform files:

module-example/vars.tf

```
variable "region" {} # the input parameters
variable "ip-range" {}
variable "cluster-size" {}
```

module-example/cluster.tf

```
# vars can be used here
resource "aws_instance" "instance-1" {...}
resource "aws_instance" "instance-2" {...}
resource "aws_instance" "instance-3" {...}
```

module-example/output.tf

```
output "aws_cluster"
value = "${aws_instance.instance-1.public_ip}, ${aws_instance.instance-2.public_ip},
${aws_instance.instance-3.public_ip}"
```

8. Use the output from the module in the main part of your code:
```
output "some-output" {
        values = "${module.module-example.aws-cluster"
}
```
9. I am just using the output resource here, but we can use the variables anywhere in the terraform code.

2.12: Terraform Commands Overview:

1. Terraform is very much focused on the resource definitions.
2. It has a limited toolset available to modify, import, create these resource definitions
    - Still, every new release there are new features coming out to make it easier to handle your resources.
    - For example, today (Q4-2016), there is still no good tool to import your non-terraform maintained infrastructure and create the definitions for you.
    - There is an external tool called terraforming that you can use for now, but it will take you quite some time to convert your current infrastructure to managed terraform infrastructure (https://github.com/dtan4/terraforming)

Command	**Description**
Terraform apply	applies state
Terraform destroy	Destroys all terraform managed state
Terraform fmt	Rewrite terraform configuration files to canonical format
Terraform get	Download and update modules

Terraform graph	create a visual representation of a configuration
Terraform import address ID	Import will try
Terraform output	Output any of your resources.
Terraform init	Initialization the terraform
Terraform plan	It will shows the infrastructure changes
Terraform push	push changes to Atlas
Terraform refresh	Refresh the remote state
Terraform remote	Configure remote state storage
Terraform show	shows the human readable output from state
Terraform state	Advanced state management
Terraform validate	Validate the terraform syntax
Terraform taint	Manually mark resources as tainted
Terraform untaint	Undo a taint

Section-3: Packer:

Packer is **an open-source DevOps tool made by Hashicorp to create images from a single JSON config file**, which helps in keeping track of its changes in the long run. This software is cross-platform compatible and can create multiple images in parallel.

- Packer is a command line tool that can build AWS AMIs based on templates.
- Instead of installing the software after booting up an instance, you can create an AMI with all the necessary software on.
- This can speed up boot times of instances
- It's a common approach when you run a horizontally scaled app layer or a cluster of something.

3.1: Terraform with Packer and Jenkins:

1. Terraform is a great fit in a DevOps minded organization
2. Tools like Terraform and Packer can be used in the Software Development Lifecycle:
    - Release, Provision and Deploy can be done using:
    - Git + Jenkins + Terraform + Packer (Amazon images)
    - Git + Jenkins + Terraform + Docker Orchestration (Docker images)

https://webme.ie/how-to-install-packer-on-ubuntu-18-04/

https://learn.hashicorp.com/tutorials/packer/get-started-install-cli

Section-4: Terraform with AWS :

4.1: Virtual Private Cloud (VPC):

1. On Amazon AWS, you have a default VPC (Virtual Private Network) created for you by AWS to launch instances in
2. Up until now we used this default VPC
3. VPC isolates the instances on a network level
    - It's like your own private network in the cloud
4. Best Practice is to always launch your instances in a VPC
    - The default VPC
    - or one you create yourself (managed by terraform)
5. There is also EC2-Classic, which is basically one big network where all AWS customers could launch their instances in
6. For smaller to medium setups, one VPC (per region) will be suitable for your needs.
7. An instance launched in one VPC can never communicate with an instance in an other VPC using their private IP addresses
    - They could communicate still, but using their public IP (not recommended)
    - You could also link 2 VPCs called peering.
8. On Amazon AWS, you start by creating your own Virtual Private Network to deploy your instances (servers) / databases in
9. This VPC uses the 10.0.0.0/16 addressing space, allowing you to use the IP addresses that start with "10.0", like this: 10.0.x.x
10. This VPC covers the eu-west-1 region, which is an Amazon AWS Region in Ireland.

Private Subnets:

Range	From	To
10.0.0.0/8	10.0.0.0	10.255.255.255
172.16.0.0/12	172.16.0.0	172.31.255.255
192.168.0.0/16	192.168.0.0	192.168.255.255

Subnet Masks:

Range	Netmask	Total Addresses	Examples
10.0.0.0/8	255.0.0.0	16,777,214	**10**.0.0.1
10.0.0.0/16	255.255.0.0	65,536	**10.0**.5.1
10.1.0.0/16	255.255.0.0	65,536	**10.1**.5.1
10.0.0.0/24	255.255.255.0	256	**10.0.0**.1
10.0.1.0/24	255.255.255.0	256	**10.0.1**.5
10.0.0.5/32	255.255.255.255	1	**10.0.0.5**

11. Every availability zone has its own public and private subnet.
12. Instances started in subnet maon-public-3 will have IP address 10.0.3.x and will be launched in the eu-west-1c availability zone (Amazon calls 1 datacenter an availability zone)
13. An instance launched in main-private-1 will have an IP address 10.0.4.x and will reside in Amazon's eu-west-1a Availability Zone (AZ)
14. All the public subnets are connected to an Internet Gateway. These instances will also have a public IP address, allowing them to be reachable from the internet.
15. Instances launched in the private subnets don't get a public IP address, so they will not be reachable from the internet.
16. Instances from main-public can reach instances from main-private, because they are all in the same VPC. This of course if you set the firewall to allow traffic from one to another.
17. Typically, you use the public subnets for internet-facing services/applications.
18. Databases, Caching services and Backends all go into the private subnets.
19. If you use a Load Balancer (LB), you will typically put the LB in the public subnets and the instances serving an application in the private subnets.

4.2: Launching EC2 instance in VPC:

1. Spinning up an EC2 instance is very straightforward.
2. Now we want to launch the instance in our newly created VPC.
    - With security groups
    - Using a keypair that will be uploaded by terraform.
3. We need a new security group for this EC2 instance.
4. A security group is a just like a firewall, managed by AWS.
5. You specify ingress (incoming) and egress (outgoing) traffic rules.
6. If you only want to access SSH (port 22), then you could create a security group that:
    - Allows ingress port 22 on IP address range 0.0.0.0/0 (all IPs)
    - It's best practice to only allow your work/home/office IP.
    - Allow all outgoing traffic from the instance to 0.0.0.0/0 (all IPs, so everywhere)
7. To be able to log in, the last step is to make sure AWS installs our public key pair on the instance
8. Our EC2 instance already refers to a aws_key_pair.mykeypair.key_name, you just need to declare it in terraform.
   keypairs.tf

   ```
   resource "aws_key_pair" "mykeypair" {
           key_name = "mykeypair"
           public_key = "${file("keys/mykeypair.pub")}"

   }
   ```
9. The Keys/mykepair.pub will be uploaded to AWS and will allow an instance to be launched with this public key installed on it.
10. You never upload your private key! you use your private key to login to the instance.

4.3: EBS (Elastic Block Storage):

1. The t2.micro instance with this particular AMI automatically adds 8 GB of EBS storage.
2. Some instance types have local storage on the instance itself.
    - This is called ephemeral storage
    - This type of storage is always lost when the instance terminates.
3. The 8 GB EBS root volume storage that comes with the instance is also set to be automatically removed when the instance is terminated.

    - You could still instruct AWS not to do so, but that would be counter-intuitive (anti-pattern)
4. In most cases the 8 GB for the OS (root block devices) suffices.
5. In our next example I am adding an extra EBS storage volume
    - Extra volumes can be used for the log files, any real data that is put on the instance.
    - That data will be persisted until you instruct AWS to remove it.
6. EBS storage can be added using a terraform resource and then attached to our instance.
7. In the previous example we added an extra volume
8. The root volume of 8 GB still exists
9. In you want to increase the storage or type of the root volume, you can use root_block_device within the aws_instance resource.

4.4: Userdata:

1. Userdata in AWS can be used to do any customization at launch:
    - You can install extra software.
    - Prepare the instance to join a cluster.
    - E.g. Consul cluster, ECS cluster
    - Execute commands / scripts
    - Mount volumes
2. Userdata is only executed at the creation of the instance, not when the instance reboots.
3. Terraform allows you to add Userdata to the aws_instance resource.
    - Just as a string
    - Using templates
4. It will install an OpenVPN application servers at boot time.

4.5: Static IPs , EIPs and Route53 :

1. Private IP addresses will be auto-assigned to EC2 instances.
2. Every subnet within the VPS has its own range (e.g. 10.0.1.0 - 10.0.1.255)
3. By specifying the private IP, you can make sure the EC2 instance always uses the same IP address.
4. To use a public IP address, you can use EIPs (Elastic IP addresses)
5. This is a public, static IP address that you can attach to your instance.
6. Typically, you will not use IP addresses, but hostnames.
7. This is where route53 comes in.

8. You can host a domain name on AWS using Route53
9. You first need to register a domain name using AWS or any accredited registrar.
10. You can then create a zone in Route53 (e.g. example.com) and add DNS records (e.g. server1.example.com)
11. Adding a zone and records can be done in terraform.
12. Rote53 has a lot of nameservers. To know your nameservers for your particular domain, you can use the output resource to output the property aws_route53_zone.example-com.name_servers

4.6: RDS (Relational Database Services):

1. RDS stands for Relational Database Services
2. It's a managed database solution:
    - You can easily setup replication (high availability)
    - Automated snapshots (for backups)
    - Automated security updates
    - Easy instance replacement (for vertical scaling)
3. Supported databases are:
    - MySQL
    - MariaDB
    - PostgreSQL
    - MicrosoftSQL
    - Oracle
4. Steps to create an RDS instance:
    - Create a **subnet group**
        - Allows you to specify in what subnets the database will be in.
    - Create a **Parameter Group**
        - Allows you to specify parameters to change settings in the database.
    - Create a **Security group** that allows incoming traffic to the RDS instance.
    - Create an **RDS instances** itself.

4.7: IAM (Identity and Access Management):

1. IAM in AWS is Identity & Access Management
2. It's a service that helps you control access to your AWS resources.
3. In AWS you can create:

    - Groups
    - Users
    - Roles
4. Users can have groups
    - for instance, an "Administrators" groups can give admin privileges to users.
5. Users can authenticate
    - Using a login / password
    - Optionally using a token: MFA / Google Authenticator
    - An access key and secret key (the API keys)
6. Roles can give users / services (temporary) access that they normally wouldn't have.
7. The roles can be for instance attached to EC2 instances.
    - From that instance, a user or service can obtain access credentials.
    - Using those access credentials, the user or service can assume the role, which gives them permission to do something.
8. An example:
    - You create a role mybucket-access and assign the role to an EC2 instance at boot time.
    - You give the role the permissions to read and write items in "mybucket"
    - When you log in, you can now assume this mybucket-access role, without using your own credentials - you will be given temporary access credentials which just look like normal user credentials.
    - You can now read and write items in "mybucket".
9. Instead of a user using aws-cli, a service also assumes a role.
10. The service needs to implement the AWS SDK
11. When trying to access the S3 bucket, an API call to AWS will occur.
12. If roles are configured for this EC2 instance, the AWS API will give temporary access keys which can be used to assume this role.
13. After that, the SDK can be used just like when you would have normal credentials.
14. This really happens in the background, and you don't see much of it.
15. IAM roles only work on EC2 instances, and not for instance outside AWS.
16. The temporary access credentials also need to be renewed, they are only valid for a predefined amount of time.
    - This is also something the AWS SDK will take care of.
17. To create an IAM administrators' group in AWS, you can create the group and attach the AWS managed Administrator policy to it.
18. You can also create your own custom policy.

4.8: Autoscaling:

1. In AWS autoscaling groups can be created to automatically add/remove instances when certain thresholds are reached.
    - e.g. your application layer can be scaled out when you have more visitors.
2. To set up autoscaling in AWS you need to setup at least 2 resources:
    - An AWS launch configuration
    - An autoscaling group
3. Once the autoscaling group is setup, you can create autoscaling policies.
    - A policy is triggered based on a threshold (CloudWatch Alarm)
    - An adjustment will be executed.
4. First the launch configuration and the autoscaling group needs to be created.
5. To create a policy, you need a aws_autoscaling_policy
6. Then, you can create a CloudWatch alarm which will trigger the autoscaling policy.
7. If you want to receive an alert (e.g. email) when autoscaling is invoked, need to create a SNS topic (Simple Notification Service)
8. That SNS topic needs to be attached to the autoscaling group.

4.9: ELB (Elastic Load Balancers):

1. Now that you have autoscaled instances, you might want to put a load balancer in front of it.
2. The AWS Elastic Load Balancer (ELB) automatically distributes incoming traffic across multiple EC2 instances.
    - The ELB itself scales when you receive more traffic.
    - The ELB will health check your instances.
    - If an instance fails its health check, no traffic will be sent to it.
    - If a new instance is added by the autoscaling group, the ELB will automatically add the new instances and will start health checking it.
3. The ELB can also be used as SSL terminator.
    - It can offload the encryption away from the EC2 instances.
    - AWS can even manage the SSL certificates for you.
4. ELBs can be spread over multiple Availability Zones for higher fault tolerance.
5. You will in general achieve higher levels of fault tolerance with an ELB routing the traffic for your application.
6. ELB is comparable to a nginx/haproxy, but then provided as a service.
7. AWS provides 2 different types of load balancers:
    - The Classic Load Balancer (ELB)
        - Routes traffic based on network information.
        - E.g. Forwards all traffic from port 80 (HTTP) to port 8080 (application)

- The application Load Balancer (ALB)
    - Routes traffic based on application level information.
    - E.g. can route /api and /website to different EC2 instances.

4.10: Elastic Beanstalk:

1. Elastic Beanstalk is AWS's Platform as a Service (PaaS) solution.
2. It's a platform where you launch your app on without having to maintain the underlying infrastructure.
3. You are still responsible for the EC2 instances, but AWS will provide you with updates you can apply.
4. Updates can be applied manually or automatically
5. The EC2 instances run Amazon Linux.
6. Elastic Beanstalk can handle application scaling for you.
7. Underlying it uses a Load Balancer and an Autoscaling group to achieve this.
8. You can schedule scaling events or enable autoscaling based on a metric.
9. It's similar to Heroku (another PaaS Solution)
10. You can have an application running just in a few clicks using the AWS console.
11. Or using the Elastic Beanstalk resources in terraform.
12. The supported Platforms are:
    - PHP
    - Java SE, Java with Tomcat
    - .NET on Windows with IIS
    - Node.js
    - Python
    - Ruby
    - Go
    - Docker
13. When you deploy an Elastic Beanstalk environment you will get a CNAME (hostname) that you can use as endpoint
14. You can use Route53 to point your domain to that CNAME
15. Once Elastic Beanstalk is running, you can deploy your application on it using the EB command line utility.
16. It can be downloaded from:
    http://docs.aws.amazon.com/elasticbeanstalk/latest/dg/eb-cli3.html

Section-5: Advanced Terraform Usage:

5.1: Interpolation:

1. In Terraform, you can Interpolate other values, using ${....}
2. You can use simple math functions, refer to other variables, or use conditionals (if-else)
3. We have been using them already throughout the course, without naming them.
    - **Variables:** ${var.VARIABLE_NAME} refers to a variables.
    - **Resources:** ${aws_instance.name.id} (type.resource-name.attr)
    - **Data Source:** ${data.template_file.name.rendered} (data.type.resource-name.attr)
    - **Math:**
        - Add, Subtract, Multiply and Divide for float types
        - Add, Subtract, Multiply and Module for integer types.

Name	**Syntax**	**Examples**
String variable	var.name	${var.SOMETHING}
Map variable	var.MAP[“key”]	${var.AMIS[“us-east-1”]}
List variable	var.LIST, var.LIST[i]	${var.subnet[i]}
Outputs of a module	module name.output	${module.aws_vpc.vpcid}
Count Information	count.FIELD	${count.index}
Path information	path.TYPE	path.cwd(current directory) path.module(module path) path.root(root module path)
Meta information	terraform.FIELD	terraform.env

5.2: Conditionals:

1. Interpolations may contain conditionals (if-else)
2. The Syntax is :
   CONDITION ? TRUEVAL : FALSEVAL
3. For example:
   ```
   resource "aws_instance" "myinstance" {
       [....]
       count = "${var.env == "prod" ? 2:1}"
   }
   ```
4. The support operators are:
   - **Equality:** == and !=
   - **Numerical comparison:** >,<,>=,<=
   - **Boolean logic:** &&, ||, unary !

5.3: Built-in Functions:

1. You can use built-in functions in our terraform resources.
2. The functions are called with the syntax name(arg1,arg2,...) and wrapped with ${...}
   - For example ${file("mykey.pub")} would read the contents of the public key file.
3. I will go over some of the commonly used functions to get you an idea of what's available.
   - It is best to use the reference documentation when you need to use a function.
     https://www.terraform.io/docs/configuration/interpolation.html

Function	Description	Example
Basename(path)	Returns the filename(last element) of a path	Basename("/home/Edward/file.txt") Returns file.txt
Coalesce(string1,string2,…) Coalescelist(list1,list2,..)	Returns the first non-empty value Returns the first non-empty list	coalesce("","","hello") returns hello
Element(list, index)	Returns a single element from a list at the given index	element(module.vpc.public_subnets,count.index)
Format(format,args,…) Formatlist(format, args,..)	Format a string / list according to the given format.	format("server-%03d",count.index+1) returns server-001,server-002

Index(list, elem)	Finds the index of a given element in a list	index(aws_instance.foo.*.tags .Env,"prod")
Join(delim, list)	Joins a list together with a delimiter	join(",",var.AMIS) returns "ami-123.ami-456"
List(item1, item2,..)	Create a new list	join(":",list("a","b","c")) return a:b:c
Lookup(map,key,[default])	Perform a lookup on a map, using “key”.returns value representing “key”in the map	lookup(map("k","v"),"k","not found") retuns "v"
Lower(string)	Returns lowercase value of “string”	Lower(“HELLO”) returns hello
Map(key,values,..)	Returns a new map using key:value	map("k","v","k2","v2") returns: {"k"="v","k2"="v2"}
Replace(string,search,replace)	Performs a search and replace on straing	replace("aaab","a","b") returns bbbb
Split(delim,string)	Splits a string into a list	split(",","a,b,c,d") returns ["a","b","c","d"]
Substr(string,offset,length)	Extract substring from string	substr("abcde",-3,3) returns cde
Timestamp()	Returns RFC 3339 timestamp	"server started at ${timestamp()}"
Upper(string)	Returns uppercased string	upper("string)" returns STRING
Uuid()	Returns a UUID string in RFC 4122 v4 format	uuid() returns uuid value
Values(map)	Returns values of a map	values(map("k","v","k2","v2")) returns ["v","v2"]

5.4: For and For Each loops:

1. Starting from terraform 0.12 you can use for and for_each loops.
2. The for-loop features can help you to loop over variables, transform it and output it in different formats.
3. For example:
    - [for s in ["this is a", "list"] : upper(s)]
    - You can loop over a list[1,2,3,4] or even a map like {"key" = "value"}
    - You can transform them by doing a calculation or a string operation.
    - Then you can output them as a list or map
4. For loops are typically used when assigning a value to an argument.
5. For Example:
    - security_groups = ["sg-12345", "sg-5678"]

        - This could be replaced by a for loop if you need to transform the input data.
    - Tags = {Name = "resource name"}
        - This is a map which can be "hardcoded" or which can be the output of a for loop.
6. For_each loops are not used when assigning a value to an argument, but rather to repeat nested blocks
7. The way to then to loop over data and output multiple literal blocks is with a foreach

5.5: Terraform Project Structure:

1. When starting with terraform on Production environments, you quickly realize that you need a decent project structure.
2. Ideally, you want to separate your development and production environments completely.
    - That way, if you always test terraform changes in development first, mistakes will be caught before they can have a production impact.
    - For complete isolation, it's best to create multiple AWS accounts and use on account for Dev and for Prod and a third one for billing.
    - Splitting out terraform in multiple projects will also reduce the resources that you will need to manage during one terraform apply.

5.6: Terraform Lock File:

1. Starting from terraform 0.14(November 2020), terraform will use a provider dependency lockfile.
2. The file is created when you enter terraform init and is called .terraform.lock.hcl
3. This file tracks the versions of providers and modules
4. The lockfile should be committed to git.
5. When committed to git, re-runs of terraform will use the same provider/module versions you used during execution.
6. Terraform will also store checksums of the archive to be able to verify the checksum.
7. Terraform will update the lockfile, when you make changes to the provider requirements.

5.7: State Manipulation:

1. The command "terraform state" can be used to manipulate your terraform state file.

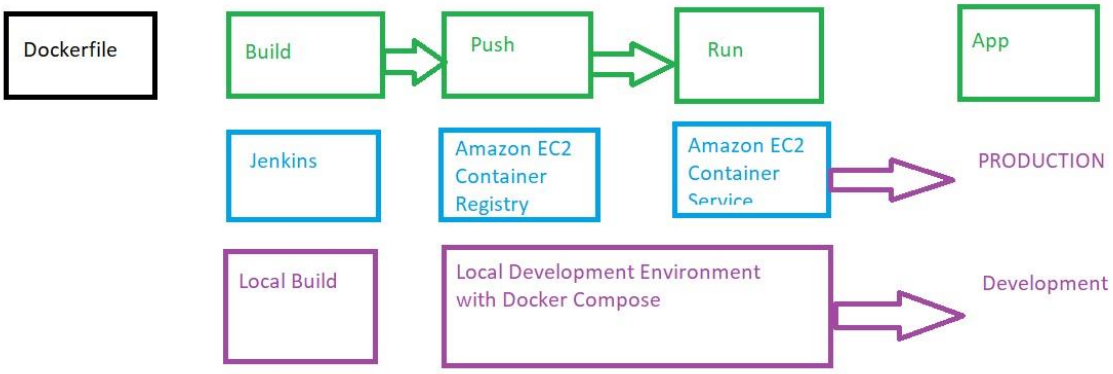
Command	Description
Terraform state list	List the State
Terraform state mv	Move an item in the state
Terraform state pull	Pull current state and output to stdout
Terraform state push	Overwrite state by pushing local file to statefile
Terraform state replace-provider	Replace a provider in the state file
Terraform state rm	Remove item from state
Terraform state show	Show item in state.

2. Here are a few use cases when you will need to modify the state:
    - When upgrading between versions, ex: 0.11 -> 0.12
    - When you want to rename a resource in terraform without recreating it.
    - When you changed a key in a for_each, but you don't want to recreate the resources.
    - Change position of a resource in a list(resource[0], resource[1],...)
    - When you want to stop managing a resource, but you don't want to destroy the resource(terraform state rm)
    - When you want to show the attributes in the state of a resource(terraform state show)

Section-6: Docker on AWS using ECS and ECR:

6.1: Introduction to Docker:

1. Just like packer builds AMIs, you can use docker to build docker images.
2. Those images can then be run on any Linux host with Docker Engine installed.



3. Using my vagrant "DevOps Box" (The ubuntu box)
4. By downloading Docker for:
    - Windows: https://docs.docker.com/engine/installation/windows/
    - MacOS: https://docs.docker.com/engine/installation/mac
    - Linux: https://docs.docker.com/engine/installation/linux
5. The demos will be done using Docker Enfgine installed in the Vagrant DevOps box (https://github.com/wardviaene/devops-box)

6.2: Building Docker Images:

1. Dockerizing a simple NodeJS app only needs a few files:
    - Dockerfile
    - index.js
    - package.json
2. To build this project, docker build can be used
3. Docker build can be executed manually or by Jenkins
4. After the docker build process you have built n image that can run the NodeJS app
5. You could already run the docker app by executing "docker run" locally
    - Docker can be run locally for development purposes
6. Instead, we are immediately going to push this image to Amazon and run this app on AWS
    - The first step will be to push this locally built images to Amazon ECR (The Elastic Container Registry-where docker images can be stored in)
    - Secondly, we will set up a docker cluster (ECS) to run our Docker applications.

7. The creation of the ECR repository can be done using terraform:

ecr.tf

```
resource "aws_ecr_repository" "myapp"{
        name = "myapp"
}
```

output.tf

```
output "myapp-repository-URL"{
        value = "${aws_ecr_repository,myapp.repository_url}"
}
```

Docker Build and Push Command

$ docker build -t myapp-repository-url/myapp

$ aws ecr get-login

$ docker push myapp-repository-url/myapp

6.3: Elastic Container Service (ECS):

1. Now that your app is dockerized and uploaded to ECR, you can start the ECS cluster.
2. ECS - EC2 container services will manage your docker containers.
3. You just need to start an autoscaling group with a custom AMI
    - The custom AMI contains the ECS agent
4. Once the ECS cluster is online, tasks and services can be started on the cluster.
5. First, the ECS cluster needs to be defined:

```
resource "aws_ecs_cluster" "example-cluster"{
        name = "example-cluster"
}
```

6. Then, an autoscaling group launches EC2 instances that will join this cluster
7. The iam role policy (aws_iam_role_policy.ecs-ec2-role-policy)
8. Before the docker app can be launched, a task definition needs to be provided.
9. The task definition describes what docker container to be run on the cluster:
    - Specifies Docker image (the Docker image in ECR)
    - Max CPU usage, Max memory usage
    - Whether containers should be linked
    - Environment variables

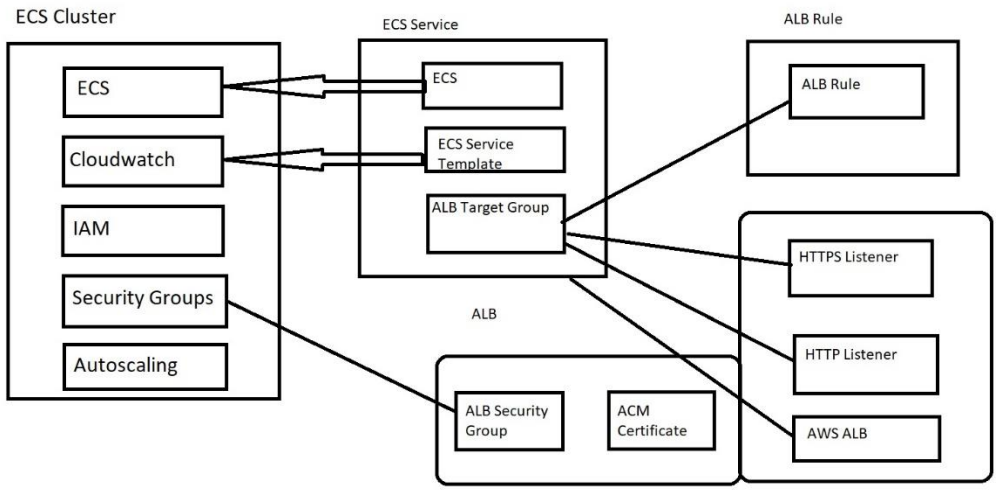
    - And any other container specific definitions
10. A service definition is going to run a specific amount of containers based on the task definition.
11. A service is always running, if the container stops, it will be restarted.
12. A service can be scaled, you can run 1 instance of a container or multiple
13. You can put an Elastic Load Balancer in front of a service.
14. You typically run multiple instances of a container, spread over Availability Zones.
    - If one container fails, your load balancer stops sending traffic to it.
    - Running multiple instances with an ELB / ALB allows you to have HA.

Section-7: Module Development:

7.1: Introduction to Module Development:

1. Terraform modules are a powerful way to reuse code
2. You can either use external modules or write modules yourself.
3. External modules can help you setting up infrastructure without much effort.
    - When modules are managed by the community, you will get updates and fixes for free.
    - https://github.com/terraform-aws-modules/ lists terraform modules for AWS, maintained by the community.
4. A few popular modules are:
    - A module to create VPC resources https://github.com/terraform-aws-modules/terraform-aws-vpc
    - A module to create an Application Load Balancer https://github.com/terraform-aws-modules/terraform-aws-alb
    - A module to create a Kubernetes cluster https://github.com/terraform-aws-modules/terraform-aws-eks
5. Writing modules yourself gives you full flexibility.
6. If you maintain the module in a git repository, you can even reuse the module over multiple projects.

7.2: ECS & ALB Module:



https://github.com/in4it/terraform-modules

7.3: Advanced Module Development:

1. We have seen till now how to write a simple module in terraform:
    - "Variable" declares the input for your modules.
    - "output" declares the output, which in turn can be used in the root project or other modules(e.g. module.mymodule.myvalue)
    - Resources can be stored with in a module to allow easy provisioning of a logic component that consists of multiple resources
2. Starting from terraform 0.12 (released May 2019), module development can take on more complex tasks. It was the start of:
    - Better expression syntax
    - Better type system (better support for map, lists)
    - Iteration constructs (for and for_each)
    - This finally allowed developers to iterate over blocks, something that was very hard, if not impossible to do in older versions.
3. Starting from terraform 0.13 (August 2020), module improvement got another boost, with the support in modules for:
    - Count
    - Depends_on
    - For_each
4. for_each in terraform can be used to iterate over resources, just like count
5. With terraform 0.13, for_each and count can also be used for modules.

Section-8: AWS CodePipeline (Continuous Delivery / Deployment:

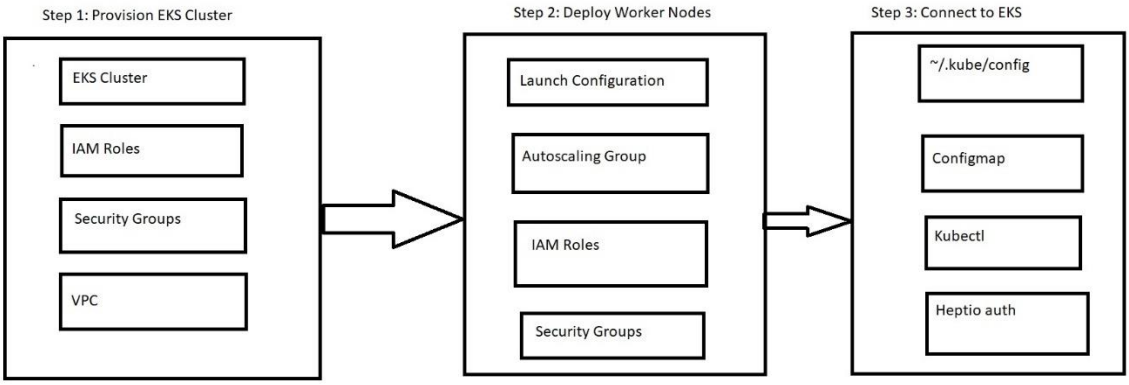
8.1: AWS CodePipeline:

1. AWS CodePipeline is a fully managed continuous delivery service.
2. CodePipeline automates the build/test/deploy pipeline
    - You can build/test applications with the typical build tools (npm,maven.gradle,..)
    - And also using docker (Using docker build & push for example)
3. You can deploy on EC2 / On-Prem / Lambda / ECS
4. You can also integrate it with Jenkins
5. AWS CodePipeline integrates with:
    - **CodeCommit**: git repository
    - **CodeBuild**: Launches EC2 instance to run your test / build phase. Can also do docker builds.
    - **CodeDeploy**: Deploys on EC2 / Lambda / ECS
6. Our setup will use **CodeCommit + CodeBuild** to build NodeJS docker image with the application code bundled + **CodeDeploy** to deploy on ECS

8.2: AWS EKS:

1. Amazon Elastic Container Service for Kubernetes is a highly available, scalable and secure Kubernetes Service.
2. It's General available since June 2018
3. Kubernetes is an alternative to AWS ECS
    - ECS is AWS-specific, whereas Kubernetes can run on anu public cloud provider or even on-premises
    - They are both great solutions to Orchestrate containers.
4. AWS EKS provides managed Kubernetes master nodes
    - There is no master nodes to manage
    - The master nodes are multi-AZ to provide redundancy
    - The Master nodes will scale automatically when necessary
    - Secure by default: EKS Integrates with IAM.
5. AWS charges money to run an EKS cluster
    - For smaller setups, ECS is cheaper
6. Kubernetes is much more popular than ECS, so if you are planning to deploy on more cloud providers / on-prem, it is a more natural choice.

7. Kubernetes has more features, but it also much more complicates than ECS- to deploy simpler apps/solutions
8. ECS has very tight integration with other AWS services, but it is expected that EKS will also be tightly integrated over time.
9.



Section-9: HashiCorp Certification:

9.1: Introduction:

1. Go to HashiCorp site https://www.hashicorp.com/
2. Select resources → select the Certifications https://www.hashicorp.com/certification

3. Select the Terraform Associate and you will get the complete information. https://www.hashicorp.com/certification/terraform-associate

9.2: Infrastructure as Code (Iac):

1. Infrastructure As Code:
    - Instead of going through the UI, write code to create resources.
    - Code will be stored in git or other VCS

    - Audit Log
    - Ability to have a review process (PRs)
    - Code can be used within a Disaster Recovery process
    - Reusability of code
    - Possible automation of provisioning
2. Terraform applies IaC using HCL (Hashicorp Configuration Language)
3. Terraform can run an execution plan to show you how the described code differs from what is actually provisioned.
4. Terraform can resolve dependencies for you, it reads all your ***.tf** files at once and creates a "Resource Graph" to know what resource should be created before another resource.
5. You know exactly what terraform will apply, using the plan and apply workflow.
    - Terraform will only update resources that need to be changed.

9.3: Terraform Basics:

1. Terraform installation has been covered in the beginning of the course.
2. Terraform uses providers, which are shipped separately with their own version numbering.
3. The terraform core contains the language interpreter, the CLI and how to interact with those providers (not the providers itself.)
4. It does not contain the code to interact with the API of the cloud providers to create resources, that code will be found in the "providers", which will be installed separately when invoking "terraform init"
5. The terraform registry is the main directory for providers and can be found at https://registry.terraform.io/browse/providers
6. The most known providers are:
    - **Cloud providers**: AWS. Azure, GCP, Oracle Cloud, Alibaba Cloud
    - Kubernetes, Helm
    - Active Directory, DNS, HTTP
    - Hashicorp Vault
7. You can immediately start using terraform resources from a specific provider (e.g aws_instance) and **terraform init** will install this provider.
8. Terraform by default will download the latest available version of that provider.
9. If you want, and it's good practice, you can specify the provider requirements in the terraform block.
10. Besides the terraform provider requirements, you can also specify the minimum terraform version.
11. Terraform will release new breaking changes between 0.12, 0.13, 0.14, ..etc.
12. Bugfixes are performed in the patch releases: 0.12.1,0.12.2, etc
13. Terraform provider versioning follows semantic versioning:

- MAJOR.MINOR.PATCH
- PATCH = bug fixes only
- MINOR = new features
- MAJOR = possible breaking changes

14. In this course we went over multiple ways of provisioning VMs:
    - Local-provisioner
    - Remote-provisioner
    - Packer
    - Cloud init
15. Provisioners (local-exec / remote-exec) are separate flows that cannot be fully controlled by terraform
    - Provisioners add a considerable amount of complexity and uncertainty
    - More coordination required: security groups need to be open, network access to the instances to run provisioning
16. Therefore, you should only use provisioners as last resort, when other approaches are not possible.
17. For most use cases, you will be able to use **Cloud init**.
    - Cloud init(user_data in aws)instance) will run after the EC2 instance will launch for the first time.
    - Other cloud providers have a similar approach (Google Cloud has metadata, Microsoft Azure custom_data, etc)
18. Since Kubernetes and other container orchestrators like ECS are used for provisioning, instance provisioning becomes less of an issue.
    - Provisioning happens when building the container, then the container is launched on a container platform.

9.4: Terraform CLI:

1. For the certification, you need to know about a few CLI commands (besides init/plan/apply).
2. **Commands:**
    - Terraform fmt
    - Terraform taint
    - Terraform import
    - Terraform workspace
    - Terraform state
3. Terraform starts with a single workspace "default"
4. You can create a new workspace using "**terraform workspace new**"
   $ terraform workspace new mytestworkspace

5. Switching to another workspace (or back to default) can be done with "**terraform workspace select name-of-workspace"**
6. Once you are in new workspace, you will have an "empty" state.
7. Your previous state is till accessible if you select the "default" workspace again.
8. When you run terraform apply in your new workspace you will be able to re-create all the resources and those resources will be managed by this new state in this new workspace.
9. This can be useful if you for example, want to test something in your code without making changes to your existing resources, for example create a new instance with encrypted root devices in a new workspace to test whether your new code works, rather than immediately trying this on your existing resource.

9.5: Terraform Modules:

1. In this course, we covered a lot of material on modules, so let's rehearse what we learned in this lecture.
2. This is a typical module declaration:

```
module "consul" {
        source = "hashicorp/consul/aws"
        version = "0.1.0"
}
```

3. This will download a specific module version from the terraform registry
4. We can also see that the module is owned by HashiCorp, because it starts with HashiCorp/
5. You don't necessarily need to use the registry, you can also use the modules directly if you create a directory for example:

```
module "mymodule" {
    source = "./mymodule" # refers to a local path
}
```

6. Terraform will also recognize GitHub (HTTPS):

```
module "mymodule" {
        source = "https://github.com/in4it/terraform-modules"
}
```

7. And also over SSH:

```
module "mymodule" {
        source = "git@github.com:in4it/terraform-modules.git"
}
```

8. These examples work with bitbucket as well (replace Github.com in bitbucket.org)
9. Passing input:

```
module "mymodule" {
```

```
        source = "./mymodule"
        myvalue = "123"
    }
```

10. In ./mymodule/vars.tf

```
    variable "myvalue" {
    }
```

11. Getting Output:

```
    module "mymodule" {
      source = "./mymodule"
      myvalue = "123"
    }

    module "other-module" {
      public_ip = module.mymodule.instance_public_ip
    }
```

12. In ./mymodule/output.tf:

```
    output "instance_public_ip" {
      value = aws_instance.myinstace.public_ip

    }
```

13. In a module you can only use the variables that are declared within that module.
14. In the root module (the root project), you can only access parameters that are defined as output in that module.
15. To access data from the root module or other modules, you can use inputs to pass information to the module.
16. To provide data to the root module, you can use outputs to pass information to the root module.

9.6: Terraform Module Registry:

1. When using modules and also providers, you can specify a version constraint

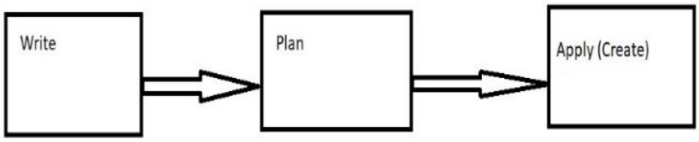
```
    version = ">=1.2.0, <2.0.0"
```

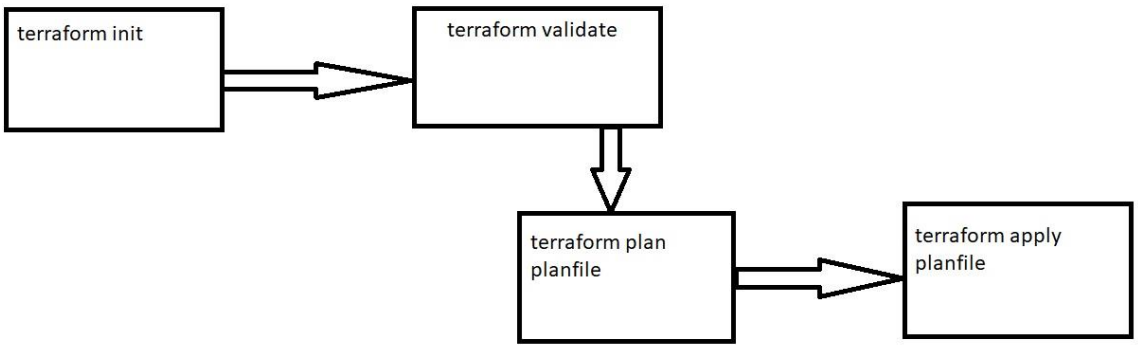
2. This version allows every version greater or equal then 1.2.0 but needs to be less than 2.0.0
3. You can separate conditions with a comma
4. The version numbering should follow semantic versioning (major.minor.patch)
5. The following operators can be used with version conditions:

- = : Exactly one version
- != : Excludes as exact version
- >,>=,<,<=
- ~>: Allows right most version to increment
- Ex: "~> 1.2.3" will match 1.2.4,1.2.5 but no 1.3.0

6. Best Practices:
    - Terraform documentation recommends to use specific versions for third party modules
    - For modules within your organization, you can use a range, for example "~> 1.2.0" to avoid big changes when you bump to 1.3.0.
    - Within modules, you should supply a minimum terraform core version to ensure compatibility
    - For providers you can use the ~> constraint to set lower and upper bound.

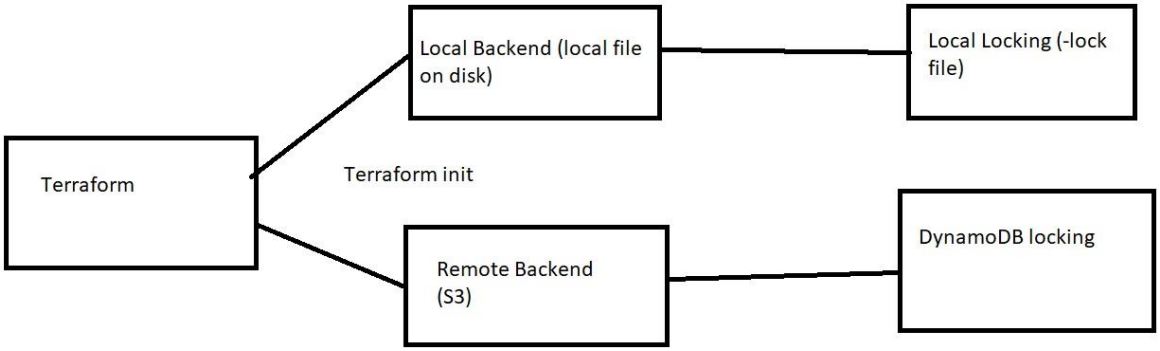
9.7: Terraform Workflow:





9.8: Implement and Maintain State:

1. The default backend in terraform is the local backend, this requires no configuration.
2. A terraform.tfstate file will be written to your project folder.
    - This is where the state is stored.
    - Every time you run terraform apply, the state will be changed and the file will be updated.
3. Once you start working in a team, you are going to want to use a remote backend.
4. Working with a remote state has benefits:
    - You can easily work in a team, as the state is separate from the code (alternatively, you would have to commit the state to version control - which is far from ideal if you need to work in a team.
    - A remote backend can keep sensitive information off disk.
    - S3 supports encryption at rest, authentication and authorization, which protects your state file much more than having it on you disk / version control/
    - **Remote Operations**: terraform apply can run for a long time in bigger projects. Backends, like the "remote" backend, supports remote operations that are executed fully remote, so that the whole operation runs asynchronously. You don’t need to be connected / keep your laptop running during the terraform apply.



5. State locking ensures nobody can write to the state at the same time.
6. Sometimes, when terraform crashes or a users internet connection breaks during terraform apply, the lock will stay.
7. "Terraform force-unlock <id>" can be used to force unlock the state, in case there is a lock, but nobody is running terraform apply.
    - This command will not touch the state, it will just remove the lock file, so it's safe, as long as nobody is really still doing an apply.
8. There is also an option -lock=false that can be passed to terraform apply, which will not use the lock file. This is discouraged and should only be used when your locking mechanism is not working.
9. Supported standard backends:
    - Artifactory (artifact storage software)
    - Azurerm (azure)
    - Consul (Hashicorp key value store)
    - Cos (Tencent cloud)
    - Etcd, Etcdv3 (similar to consul)
    - Gcs (Google Cloud)
    - http
    - Kubernetes
    - Manta (also object storage)
    - OSS (Alibaba cloud storage)
    - pg (postgress)
    - S3
    - Swift (Openstack blob storage)
10. Every backend will also have a specific authentication method
11. The configuration is done within the terraform {} block:

```
terraform {
  backend "azurerm" {
}
}

terraform {
```

```
  backend "s3" {
}
}
```

12. You can have a partial backend configuration, where you leave away some of the information.
13. This can be useful if you would like to use different backends when executing the code
14. This is often then scripted with shell scripts that call terraform with the correct arguments - this to avoid having to do this manually every time.
15. Most commonly this is used to avoid having to hardcode secrets in the terraform files, which would end up in version control.
16. There are 3 ways to pass this backend information:
    - Interactively, when the information is missing, terraform init will ask for it
    - A file
    - Key / Value pairs

    $ terraform init -backend-config = path-to file

    $ terraform init -backend-config = "bucket=mubucket" -backend-config = "otherkey = othervalue"

17. If at some point you'd like to update your state file to reflect the "actual" stat of your infrastructure, but you don't want to run terraform apply, you can run "**terraform refresh**"
18. Terraform refresh will look at your infrastructure that has been applied and will update your state file to reflect any changes
19. It will modify your infrastructure, it will only update your state file.
20. This is often useful if you have outputs that need to be refreshed or something changed outside terraform and you need to make terraform aware of it without having to run an apply.
21. You need to be aware that secrets can be stored in your state file.
    - For example, when you create a database, the initial database password will be in the state file.
22. If you have a remote state, then locally it will not be stored on disk
    - As a result, storing state remote can increase security.
23. Make sure your remote state backend is protected sufficiently.
    - For example for S3, make sure only terraform administrators have access to this bucket, enable encryption at rest. Also make sure that for every backend TLS is used when communicating with the backend.

9.9: Read, Generate and Modify Configuration (Input / Output / Local Variables):

1. There are 3 types of variables in terraform:
    - Input variables
      variable "variable name" {...}
    - Output variables
      output "variable name" {....}
    - Local variables
      locals {...}
        - They are like a temporary variables that you can use
        - Used for calculations, concatenations, conditionals where the result is later used within the resources.
2. Input variables can have the following optional arguments:
    - default
    - type
    - description
    - validation
    - sensitive
3. Type Constraints:
    - string
    - number
    - bool
4. Complex types:
    - list(<type>)
    - set(<type>
    - map(<type>)
    - object({ATTR NAME = <TYPE>,..})
    - tuple([<TYPE>,..])
    - "any" can be used as a type as well to indicate any type is acceptable.
5. You can also use "**sensitive**" in input variables, to prevent terraform from outputting the variable during plan and apply
6. Output variables have a mandatory argument: value
7. The following are Optional Arguments:
    - description
    - sensitive
    - depends_on
8. In rare cases you need depends_on to depend on another resource, before outputting the value
9. It works in the same way as "**depends_on**" in regular resources.
10. Local values can be useful to avoid repeating the same values
11. I find it can also help you to move some complexity away from the resource itself for readability.
12. You should only use local values in moderation, as it can be a bit harder for maintainer to figure out where the values come from.

9.10: Resources and Datasources:

1. In terraform, you can create "resources" and "datasources"
2. Datasources allow data to be fetched or computed from outside of terraform.
    - For example, an AMI list that can be filtered to extract AMI IDs or the lookup of an external VPC to retrieve the VPC ID.
3. Resources, unlike datasources, describes one or more infrastructure objects.
    - They typically create infrastructure components, like an EC2 resource, or a VPC subnet, a Database
4. If doe some reason normal dependency management by terraform doesn't work, you can force a resource to depend on another resource.
5. This can be done using the keyword "depends_on"
6. During terraform apply, terraform will:
    - **Refresh** the data sources
    - **Create resources** that exist in the *.tf files, but not in the state file
    - **Destroy resources** that exist in the state, but not in the *.tf files
    - **Update resources** that have different arguments in the *.tf files than on the cloud provider.
        - **Destroy and re-create** resources that have arguments changed that require re-creation
        - **In-place updates** are possible if the infrastructure API supports it.
7. Resources can be addressed using:
   <RESOURCE TYPE>.<NAME>.<ATTRIBUTE>
8. Data Sources can be addressed using:
   data.<RESOURCE TYPE>.<NAME>.<ATTRIBUTE>
9. Local resources can be addressed using:
   local.key_name
10. You can use functions in terraform to create all different sorts of behavior
    - Numeric functions(min,max,..)
    - String functions (formatting strings)
    - Collection functions (merging lists,maps)
    - Encoding functions(base64,json,yaml)
    - Date and time functions
    - Hash and crypto functions(uuid,SHA)
    - IP network functions (subnet calcualtions)
    - Type conversion(tolist,toamp,toset,..)

9.11: Secret Injection:

1. When you create a user (for the terraform operator) in AWS, you get long lived AWS credentials
2. In general, long lived credentials are to be avoided
3. there are multiple ways to avoid using long lived credentials
4. For AWS, you can use Federation, with AWS SSO or a third party, to avoid the need of users.
5. You will then work with "roles" rather than user, which provides you with short-lived credentials.
6. It's out of scope to discuss the implementations of these tools, have a look at AWS SSO, Okta, OneLogin, Azure AD, SAML to get an idea of the possibilities.
7. Other Cloud providers work in similar ways and offer their own ways to di federation.
8. Another solution would be to run terraform on one of your cloud providers instances.
9. This allows you to use the identities provided by the cloud provider
10. This will also avoid the need to long lived credentials, as short lived credentials will be automatically retrieved on the instance.
11. AWS works with a metadata endpoint that you can reach to get temporary credentials on an AWS EC2 instance. This is also supported within the terraform-aws-provider
12. Other cloud providers work in similar ways.
13. Another solution is the use of Hashicorp Vault
14. You can still use long-lived credentials, and store these in Hashicorp Vault
15. Vault can then issue short-lived credentials for your terraform operations
16. To make this happen, Vault will dynamically create AWS credentials.

9.12: Terraform Cloud:

1. Terraform cloud is a HashiCorp product.
2. It helps teams use terraform together.
3. Instead of running terraform from your own machine, or on your own Jenkins, terraform cloud will run terraform on their machines in a consistent and reliable environment
4. You have easy access to shared state, version control integration, secret data, access controls for approving changes to infrastructure, policy controls and other enterprise features
5. It includes a private terraform registry to share terraform modules.
6. Terraform cloud is hosted at https://app.terraform.io
7. You can create a free account for small sized teams
8. There are paid plans for medium size business
9. For large enterprises terraform has "Terraform Enterprise", which is the self-hosted version of Terraform cloud
10. Terraform Cloud workspaces are different than the local terraform workspaces

11. When you locally use workspaces, you are still in the same directory, using the same variables and credentials. The state is empty for every new workspace, but the state is just another file within the same project.
12. With terraform cloud workspaces, it is much more isolated. It’s much more like a separate "project" with its own variables, secrets, credentials and state.
13. The state also supports multiple versions, so you can see the previous state versions and how they match with a specific terraform run.
14. Terraform recommends to use workspaces in terraform cloud to split your monolithic terraform project in smaller projects, for example split out networking different apps.
15. Terraform **Sentinel** is a paid feature, available in Terraform Cloud
16. Sentinel is an embedded policy-as-code framework integrated with the other HashiCorp Enterprise products.
17. Sentinel allows administrators to write policy rules to put controls in place to protect or restrict what can be applied to the infrastructure.
18. A few use cases:
    - Only allow a subnet of aws_ami owners so only "Amazon" or "Ubuntu" images can be launched on AWS
    - Enforce that every resource needs to be tagged.
    - Disallow 0.0.0.0/0 in security group rules
    - Restrict EC2 instance types
    - Require S3 buckets to be encrypted with a KMS key
    - Allow only specific providers in terraform
    - Limit proposed monthly cost