**Terraform**

Cloud native tools –

AWS has cloudFormationTemplate (CFT). Azure has ARM.

Then why we need terraform –

1. If we use CFT or ARM then we need dump all our code in single json or yam file. Which will be tedious in case of complex infrastructure. While terraform provides modular approach, Workspaces, profiles, also we can maintain different files for let’s say variables, providers.
2. CFT or ARM will work only in respective cloud environments. Terraform has provider block where we can mention the provider and create infra for the same. So, terraform supports multiple cloud.
3. Performing dry checks is possible only in terraform.

Terraform is owned by hashicorp. Some other tools by hashicorp –

Packer – for image automation.

Consul – For cluster and service discovery.

Vault – sensitive data management.

Provider block –

This block defines provider (aws, azure etc.)

VPC Subnet IGW Security Group creation –

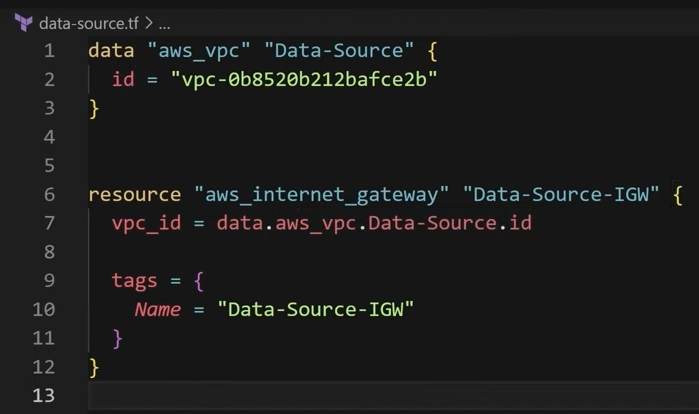
1. Create VPC



1. Create IGW
2. Create subnet
3. Create route table
4. Create route\_table-subnet association
5. Create security group

Data source –

Consider a scenario where VPC has been created manually on AWS console and we want to create and attach an IGW to that vpc using terraform. As VPC is created manually, terra is not aware of the VPC. In such scenarios we can use data source. Code snippet ->



Here, Data-source is the vpc name of the manually created vpc. We have provided the id for the same and then we can use the same to create and attach IGW to that vpc.

Explicit dependency (depends\_on) –

Let’s say we want to create NAT gateway using terraform but we want to ensure it should not be created until subnets are created. Because without subnets there is no use of NAT gateway. We can achieve this using **“depends\_on”. This is useful when Terraform does not automatically detect the correct dependency order.**

Eg. If we want to create ec2 instance, ebs volume and then attach ebs volume to ec2 instance. In that case aws\_volume\_attachment will depends\_on aws\_instance creation.

A computer screen with white text

Description automatically generated

Vars and tfvars –

When working with complex infrastructure, it is better to use variables as and when required. We will store variable names and types in vars.tf file and actual values in .tfvars file. We can access the variable in our main.tf like **var.aws\_instance\_name**. While applying terraform file we can pass the tfvars file which contains actual values. This is used when we want to deploy same infra but different configurations for different environments. In that case we can create tfvars files like dev.tfvars, prod.tfvars, qa.tfvars and then pass the respective tfvars file while terraform apply.

Create\_before\_destory –

Let’s say we have created ec2 instance and someone accidently changed key\_name from SceOps-key to SecOps-ke and then run terraform apply. In this case terraform will first delete the running instance and then create the new one. This is not ideal situation as we don’t want to delete the running instance. Here, we can use create\_before\_destroy.

A screen shot of a computer program

Description automatically generated

Here, we can use create\_before\_destroy –

A screen shot of a computer program

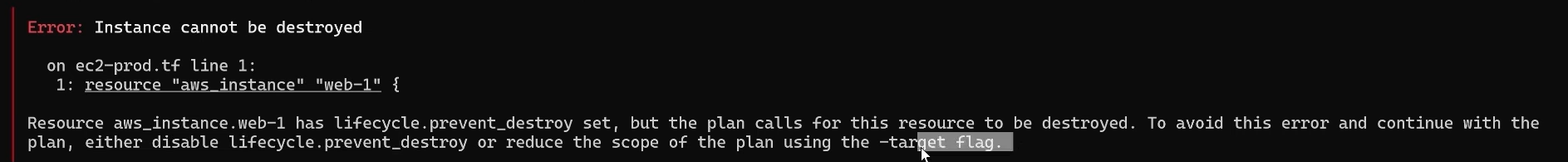
Description automatically generated

Prevent\_destory – If we want to prevent a critical resource from deleting then we can use this.

A computer screen with text

Description automatically generated

If we try to delete the resource, we will get below error –



**Terraform workspace** –

Let’s say if we want to create an infra for 3 different environments varies dev uat prod. We can use same terraform code and create different tfvars files to achieve this. But if we do so, the same state file will be maintained for all 3 environments and our infra is also gets replaced as and when we pass different tfvars file. This is not the recommended way. In such scenarios we can use terraform workspace.

Workspaces allows you to **separate your state** **and infrastructure** without changing anything in your code when you wanted the same exact code base to deploy to multiple environments without overlap. i.e. **Workspaces help to create multiple state files for set of same terraform configuration files**.

We can create 3 different workspaces using **“terraform workspace new <workspace\_name>”.**

When we create a workspace then at the remote backend 3 different state files will be stored inside the folder with workspace name.

A screenshot of a computer

Description automatically generated

Important commands –

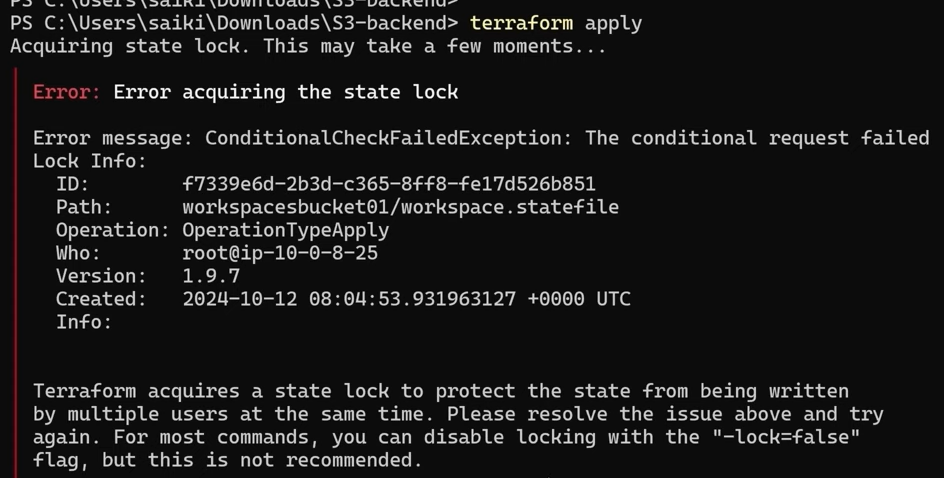
* terraform workspace new **<workspace\_name>**
* terraform workspace list
* terraform workspace select **<workspace\_name>**
* terraform workspace delete **<workspace\_name>**

Workflow –

1. create a workspace
2. switch to the workspace
3. apply the changes to deploy the infra by passing respective tfvars file.
4. Switch to different workspace
5. apply the changes to deploy the infra by passing respective tfvars file.

State locking using dynamodb –

If 2 people are applying terraform configurations at the same time then we may cause ambiguity issue. It is better to lock the state file using dynamo db locking and let the operations complete one-by-one. When someone is already deploying the changes to infra and at the same time another person hit terraform apply then we will get below error –



There might be the case where terraform stuck in locking state because of –

* A previous Terraform operation (apply, plan, destroy) was interrupted (e.g., due to a network issue, crash, or manual termination).
* Multiple users or processes try to modify the state file at the same time.
* The backend (like AWS S3 + DynamoDB or Terraform Cloud) is holding the lock and hasn’t released it properly.

If you're sure no other Terraform process is running, you can manually unlock the state:

* **terraform force-unlock <LOCK\_ID>**

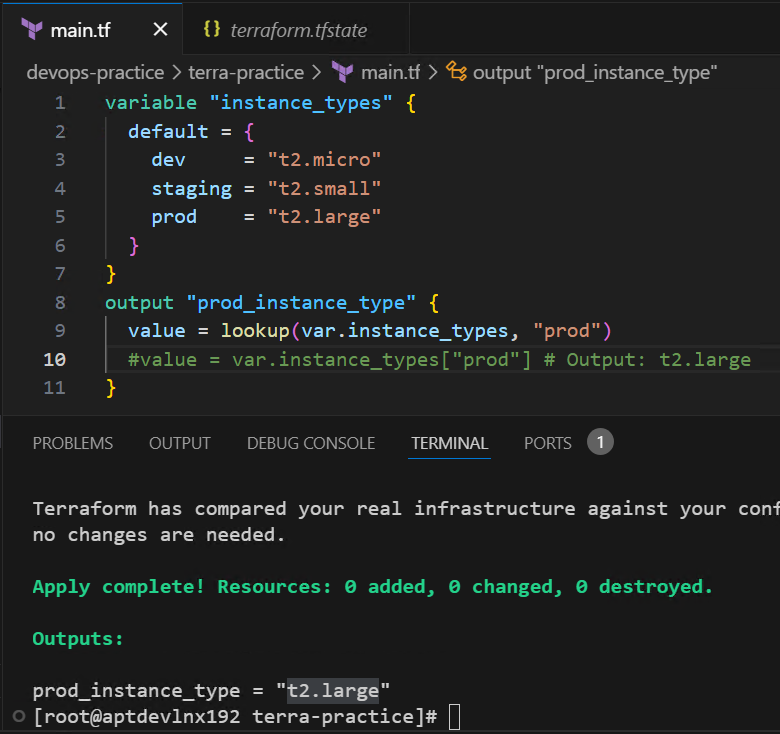
<LOCK\_ID> can be found from the error message (refer the image above).

**Functions in terraform** -

Count and index, For\_each, length, dynamic blocks, map, list, set.

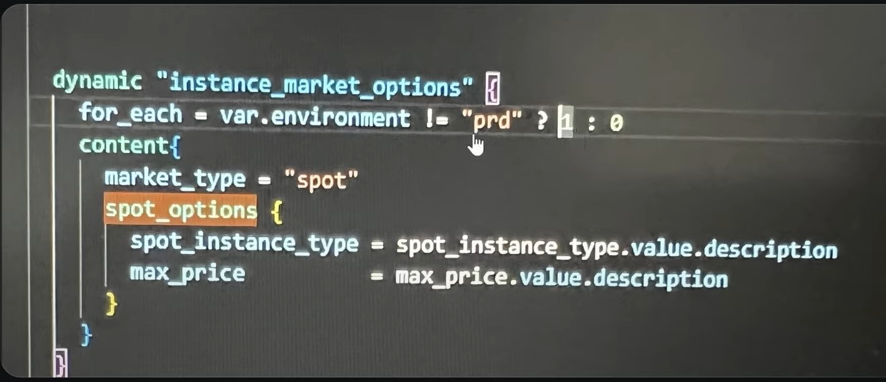


Lookup function (used for map)



Alternatively, we can use like getValue(key) in java – line no. 10 in above image.

Dynamic conditional blocks -



**Terraform provisioners** –

Scenario – We want to execute some shell commands let’s say installing nginx and starting the service on an ec2 instance. Ec2 will be created as part of terraform script and we don’t want to touch(ssh) the ec2 instance. When we run terraform apply, our terraform file copy user-data.sh to ec2 using file provisioner and should install nginx and start nginx using remote\_exec, depends\_on ec2 is created first

But the issue is, if we change any command in user-data.sh and run terraform apply again then it will recreate the ec2 instance. Which is not the recommended way.

To avoid this we can maintain provisioners in a separate file let’s say provisioners.tf

This will solve recreation issue but in this case, we need to restart ec2 instance to reflect changes done in user-data.sh

To resolve this issue we can use null\_resource and filemd5() to detect changes.

Solution –



Terraform modules –



Terraform profiles –



Terraform import, refresh and drift –

