Week3 Lab

Terraform Introduction

### Step1: Write the first configuration file with AWS provider, deploy Amazon EC2

* Change Cloud9 settings to see the hidden files

A screenshot of a computer

Description automatically generated with medium confidence

* Define the AWS provider and Amazon EC2, initialize the environment

terraform init # Initialize the Terraform environment

alias tf=terraform

tf fmt

tf validate

tf plan

tf apply

* Deploy EC2, verify that it has been deployed via aws cli and AWS management Console. Is this change disruptive?

tf fmt

tf validate

tf plan

tf apply

Graphical user interface, text, application

Description automatically generated

* Examine terraform.tfsatte file. Can you find the instance that you created there? Can you find Cloud 9 instance?

### Step2: Add tags to EC2 instance in to the Terraform configuration file.

A screenshot of a computer

Description automatically generated with medium confidence

* Run “terraform plan”, identify changes. Will the instance get redeployed?
* Deploy the changes.

tf fmt

tf validate

tf plan

tf apply

### 

### Step3: Change the instance type of the EC2 instance in to the Terraform configuration file.

Text

Description automatically generated

* Run “terraform plan”, identify changes. Will the instance get redeployed?
* Deploy the changes.

tf fmt

tf validate

tf plan

tf apply

### Step4: Add additional data storage to the EC2 instance in the Terraform configuration file.

* Add code to deploy EBC volume and volume attachment. What Availability zone should we specify?   
  See “Task 4” in the provided main.tf in Lab folder.
* Run “terraform plan”, identify changes. Will the instance get redeployed?
* Deploy the changes with “terraform apply”.

Graphical user interface, text, application, email

Description automatically generated

### Step5: Add ssh access key to the EC2 instance in the Terraform configuration file.

* Run “terraform plan”, identify changes. Will the instance get redeployed?
* Deploy the changes with terraform apply.

ssh-keygen -t rsa # Create a new SSH key pair

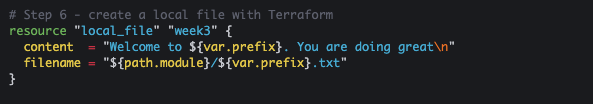
Graphical user interface, text, application, email

Description automatically generated

Why some of the changes cause instance redeployment, some cause the instance to be stopped and some do not change the state of the deployed instances

### Step6: Add [provisioning of the local file](https://registry.terraform.io/providers/hashicorp/local/latest/docs/resources/file) to the main.tf.

* Run “terraform plan”.
* Did it work? What is missing?
* Fix the problem as suggested by Terraform output and re-run “terraform plan”.
* Deploy the changes with “terraform apply”.



Content of the file:

### GContent of the file

### Step7: Examine local Terraform state

* Open the local terraform.tfstate file, note the unique ids associated with all the provisioned resources
* Apply changes to main.tf, run “terraform apply”. Did the terraform.tfstate file change?
* Remove the local file from main.tf and run “terraform apply”. Did the terraform.tfstate file change?

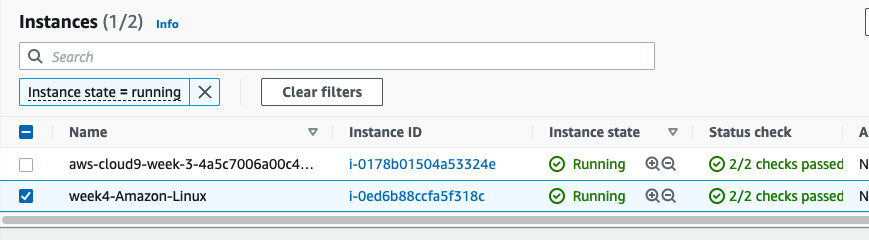
### Step8: Add variables.tf file and provide all the parameters using the options below.

Run “terraform plan” for all the provided options.

* Scalar variables and default values in variables.tf
* Map variables and default values in variables.tf
* As input flags with –var

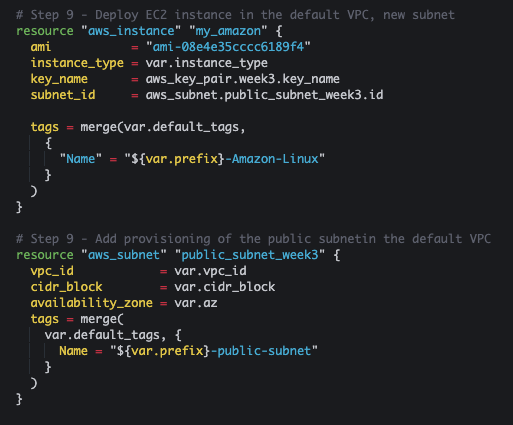


tf apply --auto-approve --var prefix="week4"



### Step9: Add provisioning of the public subnet to the main.tf and update EC2 resource block to ensure EC2 is deployed into a newly provisioned subnet.

* Is this an explicit or implicit dependency?
* Did the code work or errored out?



### Step10: Add output variables and print them out with the terraform output command

Create output variables for the instance id and subnet id in the output.tf file 

* Run “terraform plan”, identify changes. Will the instance get redeployed?

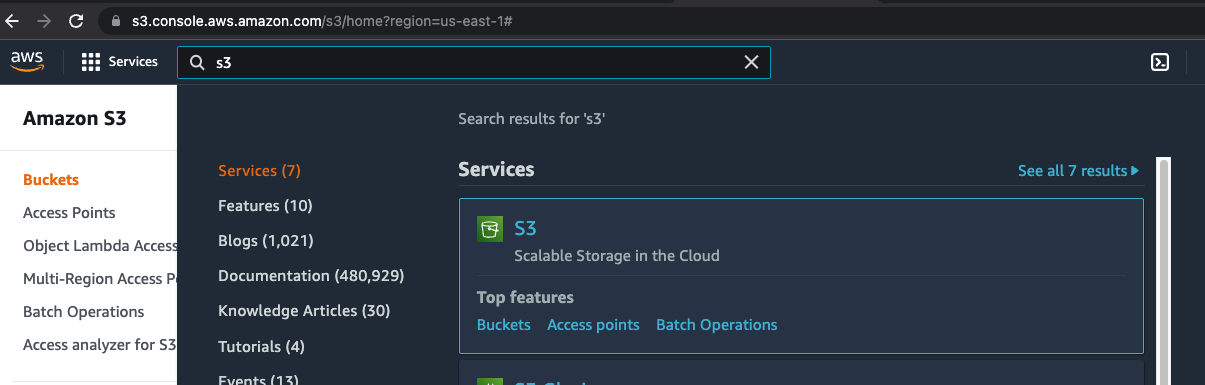


* Add public IP association
* Re-deploy the changes with “terraform apply”.
* Print out the variables with the “terraform output” command

### Terraform output

### Step11: Introducing backend block and the remote Terraform state

* Manually create an S3 bucket to hold Terraform state. To ensure the name is unique name the bucket “acs730-week3-<your Seneca username>”
* Go to S3 console



* Click on “Create bucket” and provide the bucket name. Leave all the defaults.

Graphical user interface, text, application, email

Description automatically generated

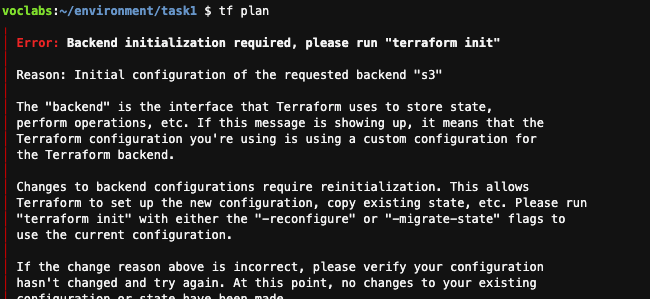
* Click “Create bucket”. You should see the bucket created as per example below.

Graphical user interface, text, application, email

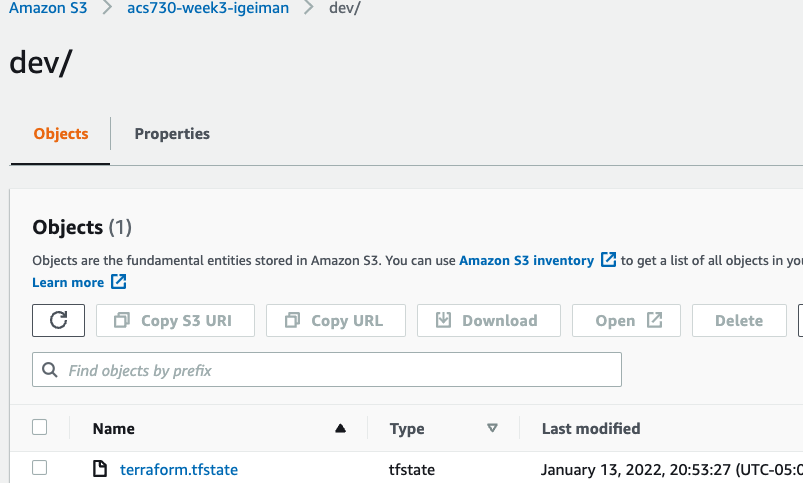
Description automatically generated

* Create a new file in your folder called config.tf
* Run terraform validate

Why did It error out? What should we do to fix the problem?



* Run terraform init
* Examine S3 bucket



* Destroy all the infrastructure with “terraform destroy”.

### Step12: Splitting the deployment into phases and working with the remote Terraform state

* Split the deployment of the subnet and EC2 instances into 2 folders:
  + 01-Networking
  + 02-Webserver
* Create 2 sub folders, leave the code related to subnet only in 01-Networking folder. Each folder should have variables.tf, main.tf, config.tf and output.tf.

Will the folders share the state file or should have different state files?

Which configuration should be deployed first?

mkdir 01-Networking

cd 01-Networking

cp \*.tf 01-Networking/

# Change the Terraform config to deploy the subnet only

tf init

tf fmt

tf validate

tf apply

Text

Description automatically generated

* Add a data block referencing state file of 01-Networking to main.tf of 02-Webserver, change the config to split the Terraform state and clean up the Terraform config from subnet deployment code

Text

Description automatically generated

* Deploy the 02-Webserver infrastructure

cd ../02-Webserver

tf validate

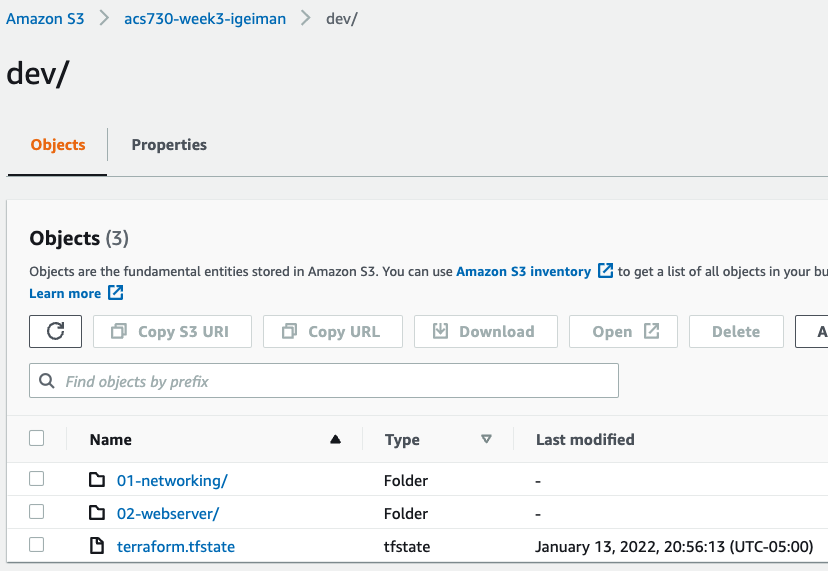
tf plan

tf apply

Graphical user interface, text

Description automatically generated

* Examine the files in S3 bucket



* Destroy the provisioned infrastructure.

What is the right order to destroy the provisioned resources?

### Step13: Creating multiple resources of the same type – provisioning subnets in different availability zones

1. Provide the list of CIDR ranges for the subnets

Graphical user interface, text, application

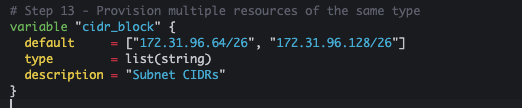
Description automatically generated

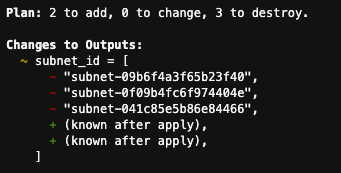
1. The number of provisioned subnets is derived from the length of the list
2. Use “count” meta-argument to provision subnets

Text

Description automatically generated

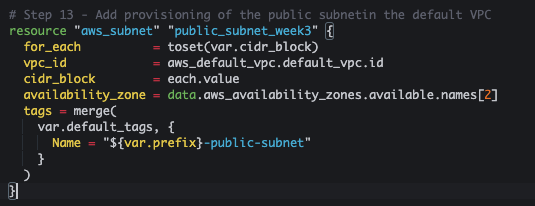
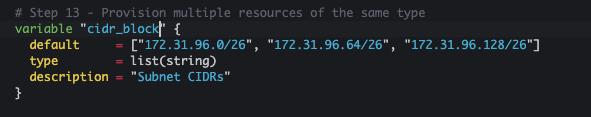
1. Remove the first element from the list of the CIDRs and run “terraform plan”. Is the result as expected?





1. Destroy the provisioned infrastructure and repeat steps 1-4 with the “for\_each” meta-argument

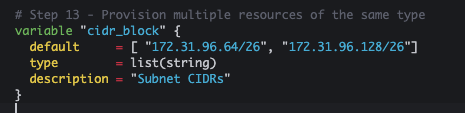
* Deploy 3 subnets



* Adjust the output

updated output

* Remove the first subnet from the list



* Run “terraform plan”.   
  Verify via AWS Console that the resources got provisioned.
* Clean up the resources with “terraform destroy”