Introduction to Terraform

Outline

- Introduction
- State
- Modules
- Best Practices
- Things to avoid
- Recap

Introduction

- Terraform is a tool for provisioning computing infrastructure.
- It supports many cloud providers. I.e. it is provider agnostic.
- It supports many resources for each provider.
- We define resources as code in Terraform templates.

Example of Terraform template

```
provider ‘aws’ {
    region = ‘us-east-1’
}

resource ‘aws-instance’ ‘example’ {
    ami = ‘ami-408c7f28’
    Instance_type = ‘t2.micro’
    Tags = { Name = ‘terraform_example’}
}
```

Terraform Providers

- It is required to have a provider defined for your terraform configuration. In this case, we are using AWS, but there are many providers that are supported by Terraform.
- A list is provided here:

  https://www.terraform.io/docs/providers/index.html

Terraform Plan

- We can use the terraform *plan* command to have Terraform show us what it is going to do before it actually executes the template.

Terraform plan

```
> terraform plan
+ aws_instance.example
    ami:                "" => "ami-408c7f28"
    instance_type:      "" => "t2.micro"
    key_name:           "" => "<computed"
    private_ip:         "" => "<computed>"
    public_ip:          "" => "<computed>"

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

Terraform apply

- We can use the terraform *apply* command to execute the terraform template and create the virtual server.

Terraform apply

```
> terraform apply
+ aws_instance.example: Creating...
    ami:                "" => "ami-408c7f28"
    instance_type:      "" => "t2.micro"
    key_name:           "" => "<computed"
    private_ip:         "" => "<computed>"
    public_ip:          "" => "<computed>"
aws_instance.example: Creation complete

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

Terraform apply

- Warning! It is not guaranteed that if terraform plan returns no errors that the terraform apply will actually work!
- If you attempt to do something that the provider doesn’t support or allow, then the apply will fail even though the plan output doesn’t report any errors!

Parameterizing Terraform templates

- We can parameterize our templates using *variables.*
- Description, default and type are optional.

```
Variable “name” {
    Description = “The name of the EC2 instance”
}
```

Parameterizing Terraform templates

- Note the use of the ${} syntax to reference var.name in tags. .

```
variable "name" {
    description = "The name of the EC2 instance"
}

resource "aws_instance" "example" {
    ami = "ami-408c7f28"
    instance_type = "t2.micro"
    tags { Name = "${var.name}" }
}
```

Submitting a value

- Note in the previous example that we didn't set a value for the variable, only its description.
- When we run this with a plan command, terraform will prompt you for the value.

```
> terraform plan
var.name
    Enter a value: foo

~ aws_instance.example
    Tags.Name: "terraform-example" => "foo"
```

Submitting a value

- We can also pass values to variables by using the -var parameter on the command line.

```
> terraform apply -var name=foo
aws_instance.example: Refreshing state...
aws_instance.example: Modifying...
tags.Name: “terraform-=example” => “foo”
aws_instance.example: Modifications complete.

Apply complete! Resources: 0 added, 1 changed, 0 destroyed.
```

Terraform variables

- Terraform supports different types of variables
  - Strings
  - Booleans
  - Lists
  - Maps

Terraform variables

- The standard way to create a variable is by simplying declaring it like so:

  variable “myvar” {

  -default = “some_value”

  -description = “some_description”

- Both the default and descriptions are optional.

Terraform strings

- Simple strings are enclosed in “ “ quotes.
- Terraform supports multiline strings like so:

```
template = <<-EOF
        #!/bin/bash
        run-microservice.sh
        EOF
}
```

Terraform arithmetic

- Terraform also supports basic arithmetic operations.
- + - * / % for integers
- + - * / for floats.

Terraform variables

- Terraform supports lists like so:

```
variable “mylist” {
    type = “list”
    default = [“foo”, “bar”, “baz”]
```

Note that type is optional. Terraform can figure out that it’s a list from the default syntax.

Terraform lists

- We can access individual elements in the list by using the *element()* function like so:

  “${element(some_list,element_num)}”

Terraform lists

- We can also access list elements directly like so:

  value = “${my_list.0}”

  This will return the zeroth element of my_list to the value variable.

Terraform lists

- We can also use wildcards.

  values = “[${my_list.*]}”

Terraform maps

- Maps are key/value pairs.
- Terraform supports maps like so:

```
variable “mylist” {
    type = “map”
    default = {“foo” =  “bar”,
                “baz” = “blech”}
}
```

Terraform functions

- Terraform has many built-in functions.
- Examples are:
  - Length: Gives a length of a list.
  - Count: Allows looping over a list.
  - Lookup: Looks up a map value based on a key.

Terraform data sources

- We can also get data directly from the cloud provider by accessing their data sources.

```
data "aws_ami" "web" {
  filter {
    name   = "state"
    values = ["available"]
  }

  filter {
    name   = "tag:Component"
    values = ["web"]
  }

  most_recent = true
}
```

Terraform data sources

- Note that in the previous example, we can declare “filters” to only get relevant data from the data source.
- Also note that the “most_recent” value is a boolean.

Terraform if statements.

- As of this date, terraform doesn’t natively support if/elif/else decision trees.
- You have to get creative.
- For example, using count with a boolean variable.
- Terraform does support the ?: ternary operator

Terraform loops

- We can use the *count* function to perform looping. For example if we have the following:

```
resource “aws_elb” “lb” {
    <params defined here>
    count = 3
}
```

- This will create three instances of this resource.

Terraform loops

- We can also loop over terraform lists:

  “${count = (length(some_list)}”
- The length function returns the number of elements in the list. Note that elements begin with zero, not one!
- We can access the current value in the count with count.index

Resource dependencies.

- We can create dependencies between resources.
- I.e. We can require that resource B requires the existence of resource A.

Resource dependencies

- Notice the use of ${} to depend on the id of the aws_instance.

```
resource “aws_eip” “example” {
    Instance  “${aws_instance.example.id}”
}

resource “aws_instance” “example” {
    ami = “ami-408c7f28”
    instance_type = “t2.micro”
    tags { Name = “${var.name}” }
}
```

Destroying resources.

- To destroy resources, use the *destroy* command.

```
> terraform destroy
aws_instance.example: Refreshing state... (ID: i-f3d58c70)
aws_elb.example: Refreshing state... (ID: example)
aws_3lb.example: Destroying...
aws_elb.example: Destruction complete
aws_instance.example: Destroying...
aws_instance.example: Destruction complete

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

Terraform states

- Terraform records the state of everything it has done.
- Terraform states are stored locally in .tfstate files by default.
- You can enable S3 storage to store your .tfstate files. The next slide shows this example.

Enabling remote state storage

- Here’s an example of enabling remote storage of terraform state files.

```
> terraform remote-config \
-backend=s3 \
-backend-config=bucket=my-s3-bucket
-backend-config=key=terraform.tfstate
-backend-config=encrypt=true
-backend-config=region=us-east-1
```

Coordinating Terraform states

- Hashicorp (Makes of Terraform) provides a service called *Atlas* which can be used to store .tfstate files. It provides file locking, but it is expensive.
- You can also create a Continuous Integration job manually with a tool like Jenkins.
- Another good alternative is to use Terragrunt.

Terragrunt

- Terragrunt is an open source wrapper for Terraform.
- Provides locking vs. DynamoDB.
- Looks for its configuration with a .terragrunt file.
- The next example shows a sample .terragrunt file.

Sample terragrunt file.

- Here’s an example of enabling remote storage of terraform state files.

```
dynamoDbLock = {
    StateFileID = “mgmt/bastion-host”
}
remoteState = {
    backend = “s3”
    backendConfigs = {
        bucket = “example-co-terraform-state”
        key = “mgmt/basion-host/terraform.tfstate”
}
```

Terragrunt

- Simply replace the terraform command with terragrunt to run commands like plan, apply and destroy.
- Terragrunt automatically obtains and releases locks on apply and destroy commands.

Example of terragrunt apply

```
> terragrunt apply
[terragrunt] Acquiring lock for bastion-host in DynamoDB
[terragrunt] Running command: terraform apply

aws_instance.example: Creating...
   ami:                          "" => "ami-0d729a60"
   instance_type:                "" => t2.micro
[...]
Apply complete!  Resources:  1 added, 0 changed, 0 destroyed.

[terragrunt] Releasing lock for basion-host in DynamoDB.
```

Terraform Modules

- Terraform modules are directories that contain one or more terraform templates.
- We can re-use these modules and templates, and subject them to version control.

Terraform Modules

- By convention we define three specific terraform templates in a module.
  - vars.tf
  - main.tf
  - outputs.tf

Vars.tf example

- vars.tf specifies module inputs.

```
variable “name” {
    description = “The name of the EC2 instance”
}

variable “ami” {
    description = “The AMI to run on the EC2 instance”
}

variable “port” {
    description = “The por tto listen on for HTTP requests”
}
```

main.tf example

- Create resources in main.tf

```
resource = “aws_instance” “example” {
    ami = “${var.ami}”
    instance_type = “t2.micro”
    user_data = “${template_file.user_data.rendered}”
    tags {Name = “${var.name}”}
}
```

Outputs.tf

- Create outputs in outputs.tf

```
Output “url” {
    Value = “http://${aws_instance.example.ip}:${var.port}”
}
```

Terraform modules

- Note that terraform modules don’t have “scope”.
- Terraform modules don’t share variables implicitly.
- You must define both inputs and outputs so that other modules can use them.
- Modules are like “functions” in other programming languages.

Terraform modules

- You always start with a ‘root’ module. The root module is the current working directory when you run ter4aform apply.
- You ‘get’ modules with terraform get.
- Module locations can be specified with the source keyword.

Terraform modules

- Module sources can be:
  - Local files
  - Git repositories
  - URL’s
  - Terraform registry locations.
  - Bitbucket
  - Mercurial repositories
  - Others...

Terraform modules example

- Here is a simple example of a module.
- Here we want to import a module “networkModule” into a new module.

```
module "networkModule" {
  source = "./module/network"
  region = “${var.region}”
}
```

Terraform modules

- Note that if you want to use variables from that other module, you must define outputs in the called module that can be used by the calling module.

Terraform provisioners

- Provisioners are used by Terraform to run remote commands on the created resource (providing that the resource supports it).
- Mainly used to run resource configuration management.
- Provisioners are added to resource definitions.

Terraform provisioners

```
resource "aws_instance" "web" {
  # ...

  provisioner "local-exec" {
    command = "echo ${self.private_ip} > file.txt"
  }
}
```

Terraform provisioners

- Note that many provisioners are remote, and will require specific information supplied in order to connect.

Terraform provisioners

Here is an example of a provisioner.

```
provisioner "file" {
  source      = "conf/myapp.conf"
  destination = "/etc/myapp.conf"

  connection {
    type     = "ssh"
    user     = "root"
    password = "${var.root_password}"
  }
}
```