clearpoint

Practical Terraform Exercises Part 2

For AWS and Azure

24 November 2022

EROAD

OVERVIEW

What we will cover

These exercises are designed to give you an understanding of some more advanced Terraform topics.

- Creating a pub sub resource
- Naming resources
- Providing input variables
- Using conditionals
- Data sourcing existing resources

Creating (another) project repository

We'll continue working in the same way as we did for workshop 1;

let's make a new repository for this workshop:

- mkdir workshop-2
- cd workshop-2
- git init
- git checkout -b main

Add a readme and make your first commit:

- Every repository should have a README.md in the root, and it's good practice to keep this up to date as you add to the repository. For this workshop, you might like to keep this up to date with any things of note you learn along the way.
- For now, let's just add a heading to with something like "#Terraform workshop 2 repo".
- git add README.md
- git commit -m "Initial commit"

Exercise 1

Setting up PubSub

Creating a pub/sub resource

Pub(lish)/Sub(scribe) is a very popular way of integrating systems without tightly coupling them.

This workshop will create resources, and iterate on them as we go through it to explain various advanced concepts about Terraform. These will all center on an instance of either an AWS Kinesis stream, or an Azure ServiceBus Topic.

To start with, we're going to setup our repository

The point of this exercise is:

- Initialise our resources
- Ensure we are set up ready for the next steps

The steps will look like this:

1. Create a directory for your root module
2. Create a ***variables.tf*** file to define configuration for our pubsub resource
3. Create a ***main.tf*** file in there with a single resource: either a Kinesis stream, or a ServiceBus topic depending on if you're using AWS or Azure
4. Create a ***providers.tf*** file to define your provider details

After this we will run your code and actually create the resources!

Workspace directory and variables

Step 1: Create a directory

For each stage of this workshop, we're going to create a directory and copy the proceeding work across. This way, you will be able to revise the progression later on if you need to.

For now, let's make a directory for our first stage (from the *workshop-2* directory):

- *mkdir -p 1-pubsub*

Step 2: Create a variables.tf file

We're going to create a variables.tf file and define just a single variable at this point. This is going to be the name of our stream/topic, and it must be unique to avoid us having naming clashes. Let's use our first and last names to try and keep things named uniquely among participants.

workshop > 2.1 > Azure > 1 - Basic ServiceBus > variables.tf >

```
1  variable "pubsub_name" {
2    type    = string
3    default = "max-powers"
4  }
5
```

Main resources

Step 3a: Create a main.tf file for AWS

This is where we will define our kinesis stream. For now, we're going to just start with the basics:

workshop > 2.1 > AWS > 1 - Basic Kinesis > main.tf > resource "aw

```
resource "aws_kinesis_stream" "main" {
  name             = "${var.pubsub_name}-stream"
  shard_count      = 1
  retention_period = 48

  stream_mode_details {
    stream_mode = "PROVISIONED"
  }
}
```

Step 3b: Create a main.tf file for Azure

This is where we will define our ServiceBus topic.

workshop > 2.1 > Azure > 1 - Basic ServiceBus > servicebus.tf > ...

```
resource "azurerm_servicebus_topic" "main" {
  name         = "${var.pubsub_name}-topic"
  namespace_id = "/subscriptions/06e10652-1077-4f95-bbfc-2b
  enable_partitioning = true
}
```

That Azure Namespace ID...

Wait; what is that extremely long Namespace ID in the Azure ServiceBus Topic?

For now, we're just going to hard code this very long namespace id, and in the next exercise, we'll learn better ways to source these values.

/subscriptions/06e10652-1077-4f95-bbfc-2b39226e13a0/resourceGroups/sandbox-rg/providers/Microsoft.ServiceBus/namespaces/sandbox-servicebus-namespace

Providers

Step 4a: Create our providers.tf file for AWS

We need to configure our provider as follows:

providers.tf M

workshop > 2.1 > AWS > 4 - Conditionals > providers.tf >

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.40.0"
    }
  }

  backend "local" {
    path = "../state/terraform.tfstate"
  }
}

provider "aws" {
  region = "ap-southeast-2"
}

```

Step 4b: Create our providers.tf file for Azure

Configure our Azure provider as follows:

providers.tf U

workshop > 2.1 > Azure > 3 - Environments > providers.tf > .

```
terraform {
  required_providers {
    azurerm = {
      source  = "hashicorp/azurerm"
      version = "~> 3.32.0"
    }
  }
  backend "local" {
    path = "../state/terraform.tfstate"
  }
}

provider "azurerm" {
  features {}
}
```

Running your code

We now have our pub/sub resources defined, and, as with our previous examples we've split our various parts of our terraform across multiple files. It's time to run these against the cloud and create the resources.

Init

Run the **terraform init** command. This will cause Terraform to fetch any providers and modules you've used. You will now find a .terraform directory containing the providers it fetched, and a lock file which pins the versions (so that subsequent runs are consistent).

Plan

Now you can run the **terraform plan** command. This will cause Terraform to analyse your code, and figure out what changes will need to be made. A detailed report is produced, as well as a summary of additions / changes / deletions.

Apply

Finally, let's run **terraform apply**. After being prompted to proceed, Terraform will provision resources and produce a report. Note that there is a lot more output here than with the first workshop, because these resources have a lot of configurable properties.

Exercise 2

Data Sources

Data sources

In our previous exercise, we created a single resource. For Azure, we had a dependency on an existing resource; a ServiceBus namespace.

Let's create a similar dependency in AWS so we can learn how to use Data Sources.

The steps will look like this:

1. Copy our code from our previous step
2. Add a new data.tf file to define our Data Sources.
3. Consume the Data Source from our main.tf

After this we will re-apply our terraform and see if anything has changed.

Prepare our directory

Step 1: Copy our previous exercise

Let’s get a head start and copy our code from our previous exercise into a new **2-datasources** directory.

- *cp -R 1-pubsub 2-datasources*

Data Sources

Step 2a: Create our data.tf file for AWS

As with our other types of objects, let's create a ***data.tf*** file for a kms key that we have stored in ssm which we can use with our kinesis stream:

data.tf M

workshop > 2.1 > AWS > 2 - DataSources > data.tf >

```
1  data "aws_ssm_parameter" "kinesis_key" {
2    name = "/workshop/kms_key/id"
3  }
```

Step 2b: Create our data.tf file for Azure

Let's define our ServiceBus Namespace data source in a ***data.tf*** file:

data.tf U

workshop > 2.1 > Azure > 2 - DataSources > data.tf > data "azurerm

```
1  data "azurerm_servicebus_namespace" "main" {
2    name                = "sandbox-servicebus-namespace"
3    resource_group_name = "sandbox-rg"
4  }
```

Consuming Data Sources

Step 3a: Consume the kms key id

Now we can consume our kms key id from our kinesis stream object in ***main.tf*** - note the additional encryption_type property too:

main.tf M

workshop > 2.1 > AWS > 2 - DataSources > main.tf > resource "aws_k

```
resource "aws_kinesis_stream" "main" {
  name             = "${var.pubsub_name}-stream"
  shard_count      = 1
  retention_period = 48

  stream_mode_details {
    stream_mode = "PROVISIONED"
  }

  encryption_type = "KMS"
  kms_key_id = data.aws_ssm_parameter.kinesis_key.value

}
```

Step 3b: Create our data.tf file for Azure

Let's get rid of that huge hardcoded string and use the id from our data source in ***main.tf***:

main.tf U

workshop > 2.1 > Azure > 2 - DataSources > main.tf > ...

```
resource "azurerm_servicebus_topic" "main" {
  name         = "${var.pubsub_name}-topic"
  namespace_id = data.azurerm_servicebus_namespace.main.id

  enable_partitioning = true
}

```

Running your code

We now have both our AWS and Azure examples consuming a Data Source. Even though you may be focusing on just AWS or Azure in this workshop, it's worth considering what you might expect to happen here for each change.

For AWS, we added in a new property to our stream object. For Azure, we replace a hard coded string for a Data Source.

Do you expect either to be changed when you run ***Plan***/***Apply***? Why?

Init

As with before, run the **terraform init** command to fetch any providers and modules you've used.

Plan

Now you can run the **terraform plan** command. This will tell us what changes terraform intends to make (if any).

Apply

Finally, let's run **terraform apply**.

Exercise 3

Environment specific configuration

Customising per environment

While we've created ourselves a single instance (each) of a pubsub resource, in a real world scenario, there would be other things to consider.

In many situations, you will want to configure your resources differently depending on the environment you are deploying it into.

For example, in dev, you might want to use the cheaper SKUs, and disable things like global replication. However in production environments, things would be different.

The steps will look like this:

1. Copy our code from our previous step
2. Add additional variables
3. Provide environment variable files
4. Consume our variables
5. Re-run our terraform to update our resource

After this we should see our pubsub resource has been changed to better reflect our environment.

Prepare our directory and add variables

Step 1: Copy our previous exercise

Let's get a head start and copy our code from our previous exercise into a new ***3-environments*** directory.

- *cp -R 2-datasources 3-environments*

Step 2: Add new variables

Now we can add new variables to our module. This time, we're not going to be providing a default, because we want the variables to be explicitly defined; we'll get to how that works next.

For now, update your variables.tf file with 2 new variables as follows (note AWS uses ***number*** as the type for message_ttl, Azure uses ***string***):

variables.tf U

workshop > 2.1 > AWS > 3 - Environments >

```
variable "pubsub_name" {
  type    = string
  default = "max-powers"
}

variable "env_name" {
  type = string
}

variable "message_ttl" {
  type = number
}
```

variables.tf U

workshop > 2.1 > Azure > 3 - Environments >

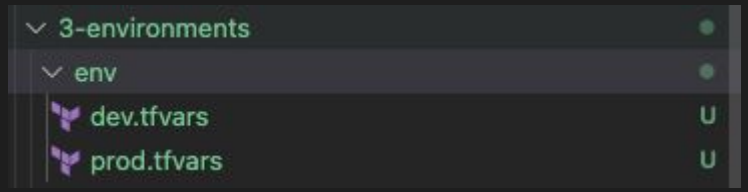
```
variable "pubsub_name" {
  type    = string
  default = "max-powers"
}

variable "env_name" {
  type = string
}

variable "message_ttl" {
  type = string
}
```

Add environment variable definitions

Step 3: Create environment variable files

We're going to be defining values for our new variable for each environment; for now, let's assume we have 2 environments - ***dev***, and ***prod***. Make a new directory in your *3-environments* directory called ***env***, and create 2 files inside the env folder named ***dev.tfvars*** and ***prod.tfvars***.

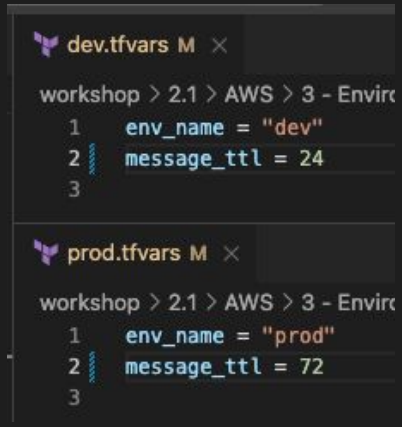
- *mkdir -p 3-environments/env*
- *touch 3-environments/env/dev.tfvars*
- *touch 3-environments/env/prod.tfvars*



Define values for variables

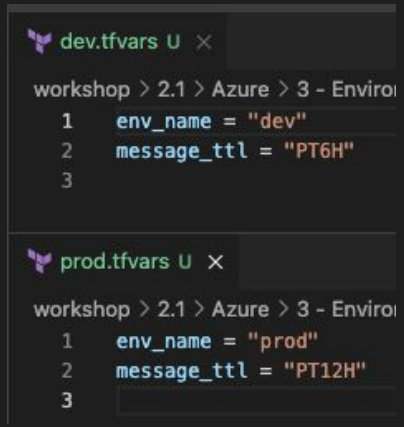
Step 4a: Define values for AWS

Edit your ***dev.tfvars*** and ***prod.tfvars*** and define values for the two variables. The message_ttl is the number of hours to retain messages. e.g.:



Step 4b: Define values for Azure

Edit your ***dev.tfvars*** and ***prod.tfvars*** and define values for the two variables. The message_ttl is in the ISO 8601 timespan format number. e.g.:



Consume variables

Step 5a: Consume variables for AWS

Update your ***main.tf*** and consume both of the two variables; one in the naming of our stream, and the other to set the retention period:

main.tf U ×

workshop > 2.1 > AWS > 3 - Environments > main.tf > resource "aws_kinesis_st

```
resource "aws_kinesis_stream" "main" {
  name             = "${var.env_name}-${var.pubsub_name}-stream"
  shard_count      = 1
  retention_period = var.message_ttl

  stream_mode_details {
    stream_mode = "PROVISIONED"
  }

  encryption_type = "KMS"
  kms_key_id = data.aws_ssm_parameter.kinesis_key.value

}
```

Step 5b: Consume variables for Azure

Update your ***main.tf*** and consume both of the two variables; one in the naming of our topic, and the other to set the default message ttl:

main.tf U ×

workshop > 2.1 > Azure > 3 - Environments > main.tf > ...

```
resource "azurerm_servicebus_topic" "main" {
  name         = "${var.env_name}-${var.pubsub_name}-topic"
  namespace_id = data.azurerm_servicebus_namespace.main.id

  enable_partitioning = true
  default_message_ttl = var.message_ttl
}
```

Running your code

We now have both our AWS and Azure examples consuming a variables that have different values per environment.

If you run the regular ***terraform init*** and ***terraform plan*** now, you'll be asked to specify the values for the variables we added.

This is because we haven't defined any default values. Instead, we now need to pass the specific environment file we want to use when we're running our plan/apply.

Init

As with before, run the **terraform init** command to fetch any providers and modules you've used.

Plan

Now you can run the **terraform plan -var-file=env/dev.tfvars** command. By specifying the dev.tfvars file, we are applying the values of the variables we want for the dev environment.

Apply

Finally, let's run **terraform apply -var-file=env/dev.tfvars**.

A quick note about local state

In our previous exercise, you'll notice that the plan/apply operations wanted to delete our existing resources.

If you ran plan/apply with the ***prod.tfvars*** file, you'll have found that the dev resources were replaced with the prod resources.

This is not how we want things to work in the real world, but this is a side effect of having a single state store on our local machines. In reality, we'd be storing state in the cloud, and we'd have separate state for each environment too.

Our infra 2.0 patterns take care of this for us automatically by appending backend configuration parameters to the ***init*** command when running our terraform from the shared terraform image.

Exercise 4

Conditionals

Conditional statements in Terraform

We now know how to vary the properties of our resources by environment, however sometimes we want to do more than just change properties in an environment. Sometimes, we want to deploy additional resources in certain environments.

We can achieve this by using conditional statements in Terraform.

Because Terraform is declarative, it's not as simple as using an ***if*** statement. Instead, we need to rely on the ***count*** parameter.

The steps will look like this:

1. Copy our code from our previous step
2. Add an additional variable
3. Update our environment variable files
4. Consume our variable
5. Re-run our terraform to update our resource

After this we should see that we can create an additional resource in the dev environment only, and not have it created in the prod environment.

Prepare our directory and add our variable

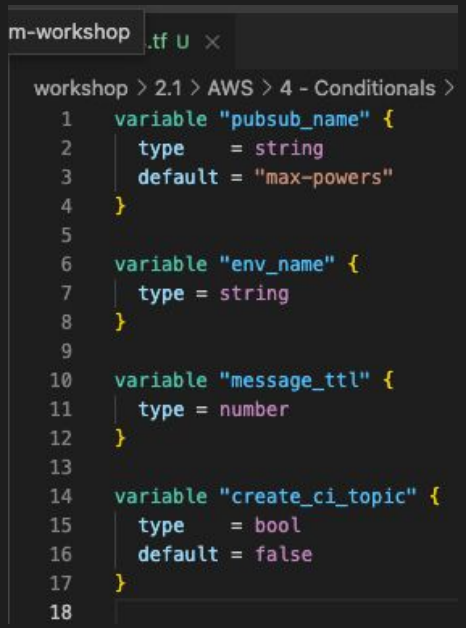
Step 1: Copy our previous exercise

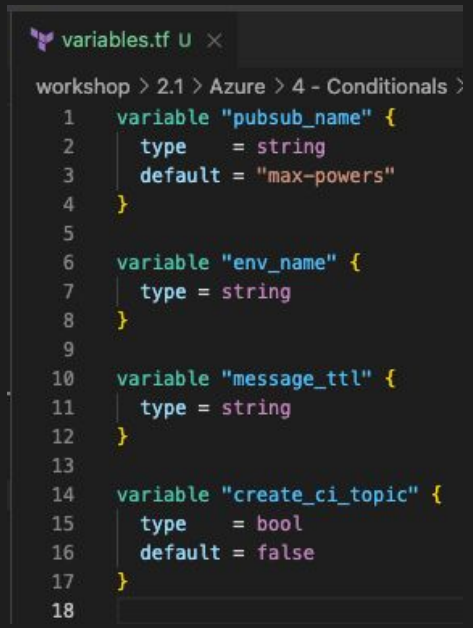
Let's get a head start and copy our code from our previous exercise into a new ***4-conditionals*** directory.

- *cp -R 3-environments 4-conditionals*

Step 2: Add new variable

Now we can add a new variable to our module. This time, we're going to use a boolean type to indicate whether we want to create a topic for our pull requests. Notice that we can define a default value for this; let's default it to false.





Define value for our new variable

Our variable ***create_ci_topic*** allows us to determine whether we should create a topic for our pull requests. The scenario here is that in our ***dev*** environment, we want an additional topic for our application to use for our pull requests that doesn't interfere with the ***dev*** environment topic.

By setting a default value to ***false***, we don't need to override this in all of our environment tfvars files. Instead, we can just set it to true in our ***dev.tfvars***.

Step 3: Define values for our variable

Edit your ***dev.tfvars*** and set the value of ***create_ci_topic*** to ***true***. You don't need to add anything to the ***prod.tfvars*** file.

Consume our boolean variable

Step 5a: Consume variable for AWS

Update your ***main.tf*** and add an additional stream. We're going to use the ***count*** property with a ternary expression to decide how many CI streams we're going to create; either 0, or 1:

```
14  resource "aws_kinesis_stream" "main" {
15    count = var.create_ci_topic ? 1 : 0
16
17    name             = "CI-${var.pubsub_name}-stream"
18    shard_count      = 1
19    retention_period = var.message_ttl
20
21    stream_mode_details {
22      stream_mode = "PROVISIONED"
23    }
24
25    encryption_type = "KMS"
26    kms_key_id = data.aws_ssm_parameter.kinesis_key.value
27  }
```

Step 5b: Consume variable for Azure

Update your ***main.tf*** and add an additional stream. We're going to use the ***count*** property with a ternary expression to decide how many CI topics we're going to create:

```
 9  resource "azurerm_servicebus_topic" "ci" {
10
11    count = var.create_ci_topic ? 1 : 0
12
13    name         = "CI-${var.pubsub_name}-topic"
14    namespace_id = data.azurerm_servicebus_namespace.main.id
15
16    enable_partitioning = true
17    default_message_ttl = var.message_ttl
18  }
```

Running your code

We now have both our AWS and Azure examples adding an optional CI stream/topic in the dev environment.

Let's run our terraform init -> plan -> apply in dev and see what happens.

Once you've run that, run terraform init -> plan for prod, and check that it doesn't try to create the CI stream/topic.

Init

As with before, run the **terraform init** command to fetch any providers and modules you've used.

Plan

Now you can run the **terraform plan -var-file=env/dev.tfvars** command. Remember, we're specifying the dev.tfvars file so we are applying the values we want in the dev environment.

Apply

Finally, let's run **terraform apply -var-file=env/dev.tfvars**.

Some more logical expressions

In this exercise, we've used the simplest form of logic in Terraform. However, there are several other logical expressions you can use.

You can use various Operators in your Terraform to compute values at runtime.

You can use For Expressions or Splat Expressions for looping through collections, and creating lists of resources.

You can also call functions to apply logic to values; have a look through the library of functions here.

With the combination of these various expressions, you are able to create quite complex terraform modules that have all kinds of rules in them to create some pretty complex infrastructure.

The Platform team makes use of a lot of the more advanced aspects of Terraform to stand up the default infrastructure for each product domain, and handle a wide range of variation across them.

If you want help with some more prickly logical scenarios, the Platform team will be able to help you!

Other points of discussion

- Accessing resources that are created using count
- Secrets from TF resources
- reading the v2 metadata for your default infra (subnet-ids, vpc-ids etc)

clearpoint

Thank you.