clearpoint

Practical Terraform Exercises Part 1

For AWS and Azure

24 November 2022

EROAD

OVERVIEW

What we will cover

These exercises are designed to quickly get you up to speed with some of the basics of Terraform.

- Setting up your environment
- Starting your project repository & designing your first root module
- Defining variables and outputs
- Implementing a module using local providers
- Running init / plans and applies
- Checking outputs
- Implementing a module with remote providers (AWS / Azure etc)
- Running against a cloud environment and checking the resources created

Setting up your environment

There are several things you will need to have before you can start to do anything with Terraform.

- GIT cli (so we can work with code repos and versioning)
- The Terraform binary
- A code editor
- Terraform plugins for your editor
- CLI credentials (key-pairs etc) for the target cloud environments you wish to work on
- For Azure you will need the cli-tool installed

These basic tools will allow you to create and manage Terraform code, and to run it against your cloud of choice.

Instructions for installing **GIT** can be found here. You will need to be able to run the **git** command from your terminal.

Terraform itself can be downloaded from here. As with GIT, you will need to be able to run the **terraform** command from your terminal.

Visual Studio Code is a high quality cross-platform code editor. You can download it for free, then install the official Hashicorp Terraform plugin for code highlighting and IDE integration.

Starting your project repository

You will need somewhere to work on your code. This will start off as being a directory, and we will turn it into a GIT repository:

- *mkdir workshop-1*
- *cd workshop-1*
- *git init*
- *git checkout -b main*

We will be accumulating some files which you *won't* want to commit. We can exclude these from git by making a ***.gitignore*** file containing the following:

- terraform.tfstate
- terraform.tfstate*
- .terraform

Add a readme and make your first commit:

- Use your editor to create a file called ***README.md*** in the root of your new repo (Markdown is a widely-supported standard for text documentation that can also render nicely in GUIs, supported by GitHub and Terraform Cloud). Something as simple as "Terraform workshop 1 repo" will do for now.
- *git add README.md .gitignore*
- *git commit -m "Initial commit"*

Exercise 1

Designing your first root module

Designing your first root module

The “root” module is Terraform’s *entrypoint* to your code, making it a good place to start.

We’ll keep the first one simple - no submodules, just one directory with some files in it. We will use the random provider to generate a password, and use outputs to view it.

The point of this exercise is:

- To experience Terraform in its most simple mode
- To see how state is stored locally
- To see how outputs work

The steps will look like this:

1. Create a directory for your root module
2. Create a ***variables.tf*** file with a variable to define the length of the password, with a sensible default value
3. Create a ***main.tf*** file in there with a single resource: a password generated using the random provider
4. Create an ***outputs.tf*** file in there with a single output: the result of the random password

After this we will run your code and see what happens! We will then be able to take our first look at a state file, and try a couple of updates.

Workspace directory and variables

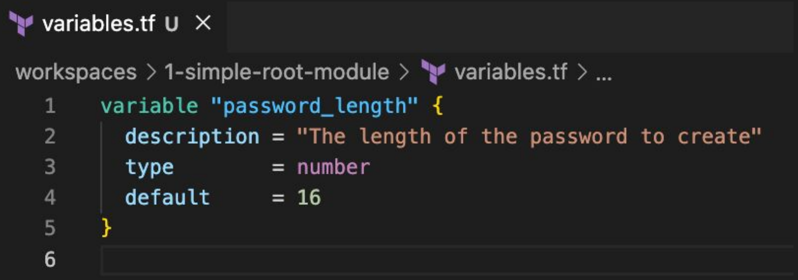
Step 1: Create a directory

We will lay your repository out as a small *monorepo*. This will allow us to host several different root modules and submodules together.

- *mkdir -p workspaces/1-simple-root-module*

Step 2: Create a variables.tf file

Here we define a single input variable which determines the length of the password to generate, with a default value.



variables.tf U ×

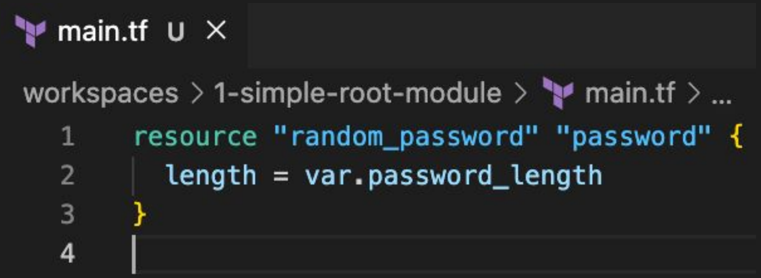
workspaces > 1-simple-root-module > variables.tf > ...

```
variable "password_length" {
  description = "The length of the password to create"
  type        = number
  default     = 16
}

```

Main resources and outputs

Step 3: Create a main.tf file

This is where we will define the random password resource. The filename "main.tf" is something like a tradition in Terraform, generally fine as long as it is relatively small (larger modules are easier to work with when the resources are logically separated into different files).

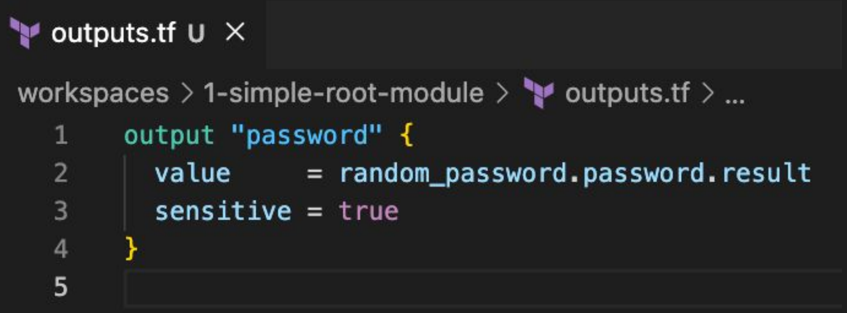

main.tf U ×

workspaces > 1-simple-root-module > main.tf > ...

```
resource "random_password" "password" {
  length = var.password_length
}

```

Step 4: Create an outputs.tf file

This is where we tell Terraform about the values that we would like to make available after the workspace has been applied.


outputs.tf U ×

workspaces > 1-simple-root-module > outputs.tf > ...

```
output "password" {
  value     = random_password.password.result
  sensitive = true
}

```

Running your code

We now have an extremely basic module ready to go. This whole thing could have been written as a single file, but it is good to get into the habit of logically separating things into different files. Terraform automatically includes every .tf file in each module, and calculates the order of execution by building up an internal dependency graph (e.g. it knows that it can't output the password until it has generated it).

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

Now the big moment - run **terraform apply**. After being prompted to proceed, Terraform will provision resources and produce a report. Note the sensitive output (a non-sensitive / insensitive output would just be displayed in-line here).

Read the output

Run **terraform output password** to show our result

Inspecting the state file

The next thing to look at is the stored state, but first try running **terraform apply** again. No changes? My infrastructure matches the configuration? How could this be? All we did was make up a random password. How does Terraform know that we have done this?

Take a look at the new file that has appeared in your directory called **terraform.tfstate** (your code editor should make this easier to read by apply JSON syntax highlighting). This is the default location for Terraform to keep track of state, and it contains all of the metadata and attributes for the resources maintained by this workspace.

Fortunately you don't have to work with this file directly - Terraform provides some useful ways to interact with the state.

- List the resources in the state with the **terraform state list** command
- View our specific password resource using the **terraform state show random_password.password** command (note how the sensitive values are obfuscated)

Now let's cause some trouble with the **terraform state rm random_password.password** command. You will no longer be able to list or show that resource. In fact Terraform has completely forgotten about it!

Now you can try running a plan / apply again. What do you expect to find?

Congratulations on your first experience of data-loss with Terraform!

Exercise 2

Provisioning resources in the cloud using modules

Provisioning resources in the cloud

Now we will look at provisioning cloud resources, and to keep things interesting we will be using modules to abstract the underlying implementation - the idea is to have a common interface that could be used to create resources on either AWS or Azure.

We have chosen to work with relatively simple cloud resources in this exercise to avoid running into resource limits in our accounts, and to avoid the types of resources that take a long time to provision and destroy. A cloud storage bucket is an obvious choice to start with, and one that you will likely be using quite often in the real world.

Storage buckets are a good example of a “universal” cloud service - nearly all cloud providers offer a storage-bucket service. Sometimes it helps to think in these generic terms.

The steps will look like this:

1. Create a root module (just like last time)
2. Create a storage module for your cloud of choice
3. Configure providers

After this we should be able to run your code and see a storage bucket created in either AWS or Azure.

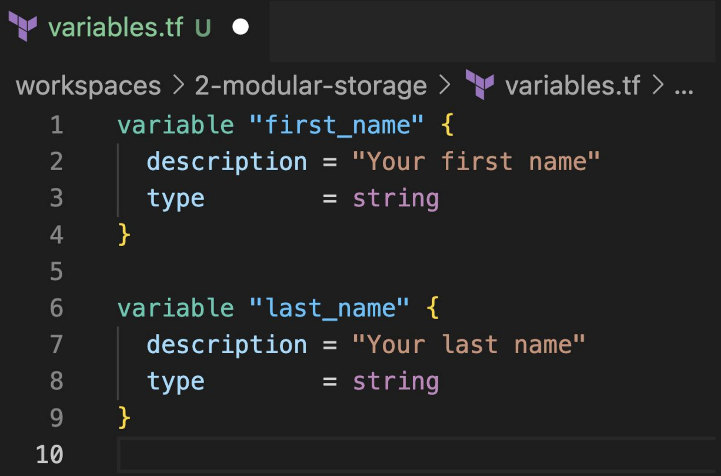
The root module

Step 1: Create a root module

As with the previous exercise, we will start with a root module.

- *mkdir -p workspaces/2-modular-storage*
- Your ***variables.tf*** file will specify 2 string variables for your first and last name (you can provide defaults, or leave it to prompt each time)
- Your ***main.tf*** file will call to a module (which doesn't exist yet)
- And your ***outputs.tf*** file will return a value from the module we will call

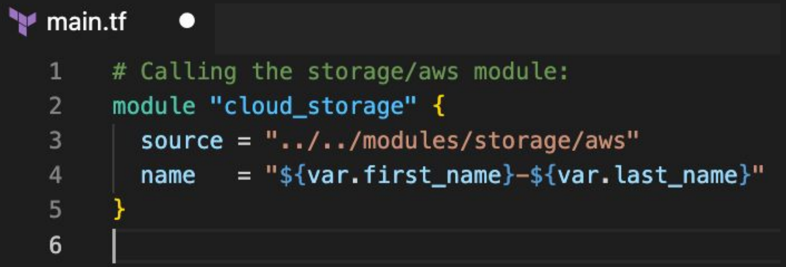
The ***variables.tf*** file with 2 string variables. Include your first and last names as defaults if you like.



```
variable "first_name" {
  description = "Your first name"
  type        = string
}

variable "last_name" {
  description = "Your last name"
  type        = string
}

```

Main.tf and outputs.tf

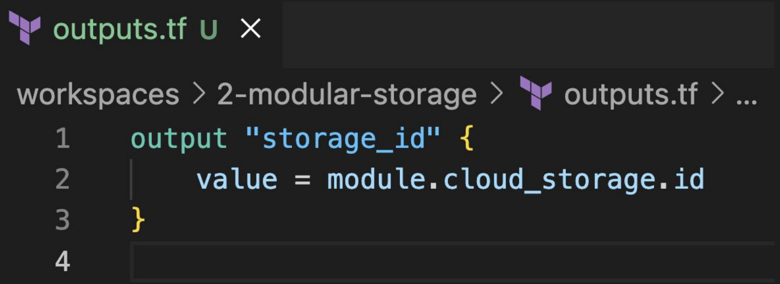
The ***main.tf*** file calling a module (which doesn't yet exist, but will soon). This is where you get to choose which cloud you will implement this module for (AWS or Azure). I've chosen AWS here, but for now it is just a name.



main.tf ●

```
# Calling the storage/aws module:
module "cloud_storage" {
  source = "../../modules/storage/aws"
  name   = "${var.first_name}-${var.last_name}"
}

```

And finally your ***outputs.tf*** file, referencing an output from the storage_bucket module.



outputs.tf U ×

workspaces > 2-modular-storage > outputs.tf > ...

```
output "storage_id" {
    value = module.cloud_storage.id
}

```

The storage module

This time instead of bundling the resources straight into the root module we will create an external module and call that from our root.

Modules are the building blocks of reusable code. Although in our example it actually takes more lines and effort, modules are generally a good way to hide some complexity and allow others to benefit from your work.

The next few steps take us through the process of defining inputs, resources, and outputs - just like we did with the root module. Everything is a module really.

As this is where we actually start to interact with cloud providers we will need to be specific with AWS and Azure, so each step shows you both ways.

Pick one (probably best), or if you're feeling complicated try both.

Links to the online Terraform documentation have been included where relevant. It is a good idea to become familiar with how to use this online documentation - it is of very high quality, and something that you will end up using extensively.

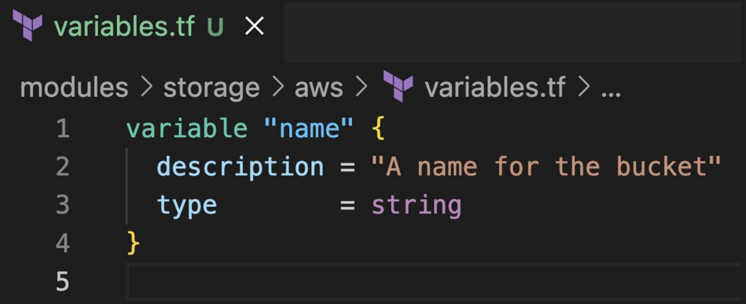
The storage module – variables

AWS

The AWS version will be implemented using the aws_s3_bucket resource from the AWS provider.

- *mkdir -p modules/storage/aws*

Let's start with the variables for this module:



variables.tf U ×

modules > storage > aws > variables.tf > ...

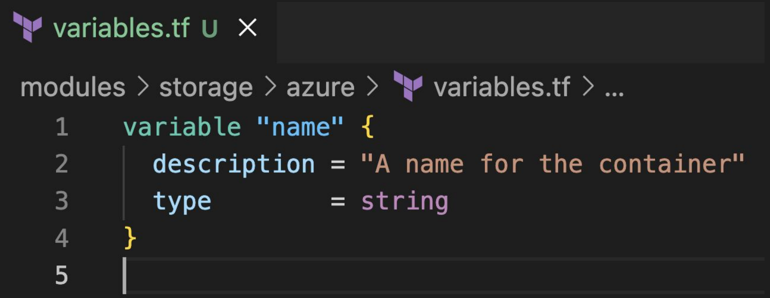
```
variable "name" {
  description = "A name for the bucket"
  type        = string
}

```

Azure

The Azure version will be implemented using the azurerm_storage_container resource from AzureRM.

- *mkdir -p modules/storage/azure*

We will use the same input variables here, for consistency:



variables.tf U ×

modules > storage > azure > variables.tf > ...

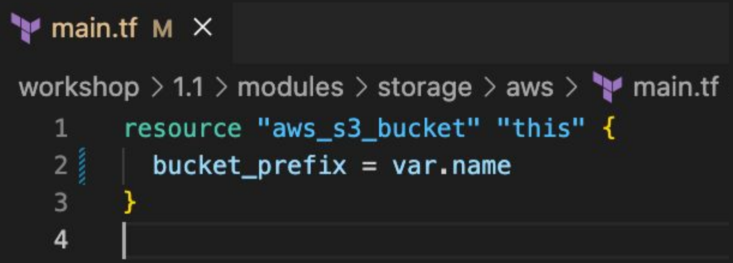
```
variable "name" {
  description = "A name for the container"
  type        = string
}

```

The storage module – resources

AWS

Now we use the input variable as the name for a new S3 bucket. This is a very simple resource!

There are of course many more things you can do with S3 buckets - read the docs for this resource to get an idea.



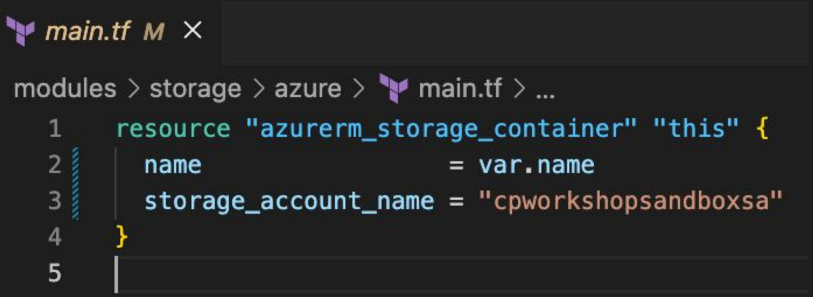
main.tf M ×

workshop > 1.1 > modules > storage > aws > main.tf

```
resource "aws_s3_bucket" "this" {
  bucket_prefix = var.name
}

```

Azure

Again we use the input variable to give a name to our azure storage container. This resource has 2 *required* arguments, so we will just specify a hard-coded one for storage_account_name (we already made this before the workshop).



main.tf M ×

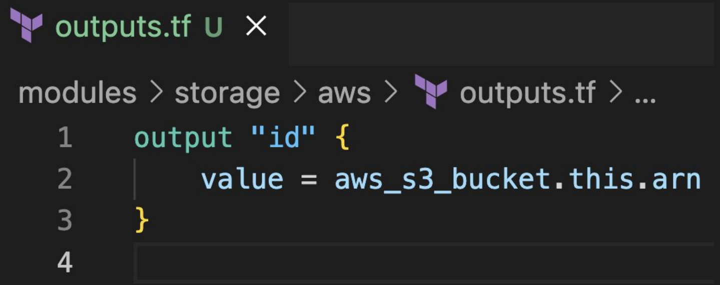
modules > storage > azure > main.tf > ...

```
resource "azurerm_storage_container" "this" {
  name                 = var.name
  storage_account_name = "cpworkshopsandboxsa"
}

```

The storage module – outputs

AWS

Here we will export the ARN attribute of the S3 bucket in the module. This is more than the ID, but in the case of S3 the ID is just the name of the bucket so is less interesting for our exercise.



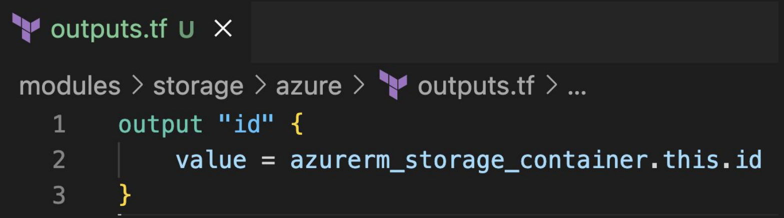
outputs.tf U

modules > storage > aws > outputs.tf > ...

```
output "id" {
    value = aws_s3_bucket.this.arn
}

```

Azure

As we kept the input variables consistent, let's also keep the outputs consistent. Here we will use the ID attribute.

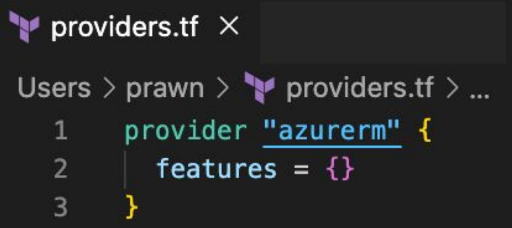


outputs.tf U

modules > storage > azure > outputs.tf > ...

```
output "id" {
    value = azurerm_storage_container.this.id
}
```

You will also need to configure a default Azurerm provider:



providers.tf

Users > prawn > providers.tf > ...

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

OK OK let’s try it out

At this point everything should be ready to go. We have a root module, and the storage module which will be called.

As with the first exercise, we need to run ***terraform init*** first (from the root module’s workspace directory). Terraform will analyse your code, and download any providers it needs. It will download the *latest* version (unless you have pinned a specific version). Take a look in the *.terraform* directory - you will find a binary of the provider (the *.terraform.lock.hcl* file will tell you exactly which version).

Now go ahead and try ***terraform plan***. Unless you’ve already set up your cloud credentials Terraform will probably complain and refuse to do anything. **It needs access to your cloud provider to proceed!**

Terraform providers can be configured in a couple of different ways.

One way is to configure them in code. This can be useful for certain arguments, but you wouldn’t want to commit credentials to a code repository.

Fortunately most Terraform providers are built with practical concerns such as this in mind, and allow you to configure them with environment variables. This is particularly useful in CI/CD pipelines, as env-vars are a standard way to inject secret values without having to store them in code.

Generally providers make use of the same environment variables as the common CLI tools, which keeps things as familiar as possible.

Configuring providers with env vars

AWS

The standard AWS env vars apply here:

- AWS_ACCESS_KEY_ID
- AWS_SECRET_ACCESS_KEY
- AWS_DEFAULT_REGION

Once these env-vars are exported you can try running ***terraform plan*** again, then ***terraform apply***.

Azure

The standard Azure env vars can be used here:

- ARM_CLIENT_ID
- ARM_SUBSCRIPTION_ID
- ARM_TENANT_ID

Once these env-vars are exported you can try running ***terraform plan*** again, then ***terraform apply***.

Azure (alternate)

You can also login with an account:

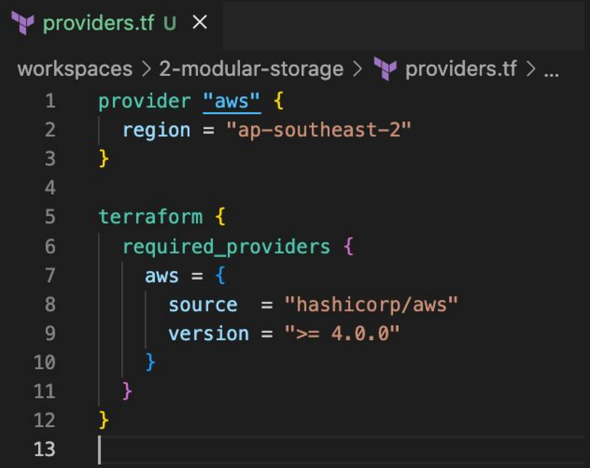
- az login (opens browser, then you login)
- az account set -s 06e10652-1077-4f95-bbfc-2b39226e13a0

Check the output value, have a look at the state file, have a go at changing parameters and doing more plans / applies. You could even try removing your resource from the state or deleting the state file, then attempting to apply again. What do you think may happen? Eventually have a go with ***terraform destroy*** to clean up your resources (unless they're now orphaned, in which case we'll make Stu clean them by hand).

Configuring providers in-code

We *do* also have the option of hard-coding some of the provider configurations. This may be particularly important when it comes to pinning *specific* versions (not just the latest one that happens to be available at the time). Try it out:

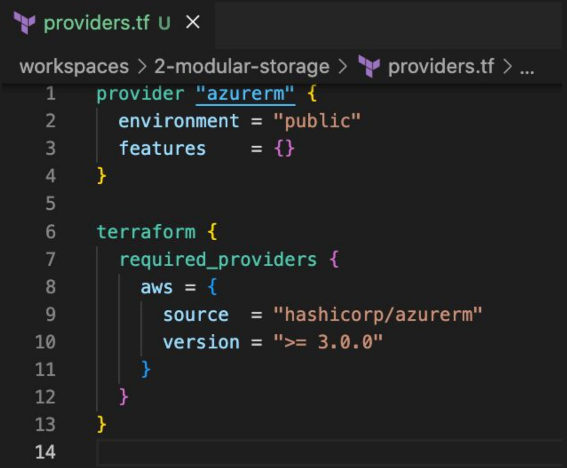
AWS


providers.tf U ×

workspaces > 2-modular-storage > providers.tf > ...

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

terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = ">= 4.0.0"
    }
  }
}
```

Azure


providers.tf U ×

workspaces > 2-modular-storage > providers.tf > ...

```hcl
provider "azurerm" {
  environment = "public"
  features    = {}
}

terraform {
  required_providers {
    aws = {
      source  = "hashicorp/azurerm"
      version = ">= 3.0.0"
    }
  }
}
```

Format (fmt)

To be a “good citizen” of a shared Terraform code repo you should always think to *format* your code correctly. The Terraform CLI tool actually comes with a formatting command which takes the uncertainty out of this:

- ***terraform fmt***

This command will analyse all of the .tf files in the current directory, and make sure that certain aspects of the format are correct. It also makes sure that there are no errors which would prevent your code from being parsed.

There are other ways to run this command as well (such as against a specific file, or recursively against every file in the tree). It is often used as a kind of *linter* step in automated Terraform pipelines.

- Go ahead and try this command against the Terraform code you’ve written so far
- Have a go at intentionally breaking a .tf file (remove a quote or something) and see what happens
- Mess around with indentation, then see what happens
- Note the non-zero return code if anything was changed or incorrect

Wrapping up

Admittedly all we did was make a random password and a cloud storage bucket / container, but all great journeys have to start somewhere.

You could easily build on top of these exercises in your own time:

- Use Terraform's template provider to render a text file containing your generated secret password (because that sounds like a secure thing to do)
- Write that file to an object in your cloud storage bucket
- Use a data source to get a list of storage buckets and return them as an output
- NO MORE CLICKOPS!

Terraform and infrastructure-as-code is a wide subject, and we've only really just begun to scratch the surface with these exercises.

Hopefully you are now feeling comfortable with the basics - the rest of your journey will build on top of this!

Please feel free to keep a copy of these slides and the code repo that goes with them for future reference. It is often useful to refer to your old code to avoid reinventing the wheel.

Also remember to commit any code you wrote during these exercises to your local repo - it is a good habit to get into.

clearpoint.

Thank you.