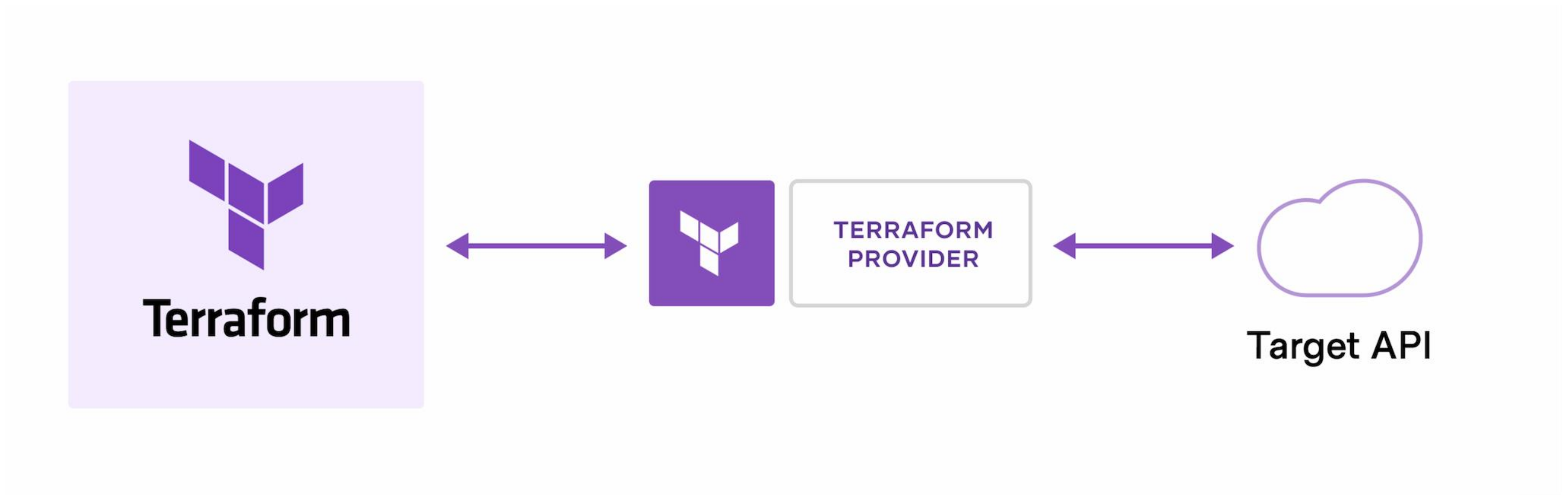
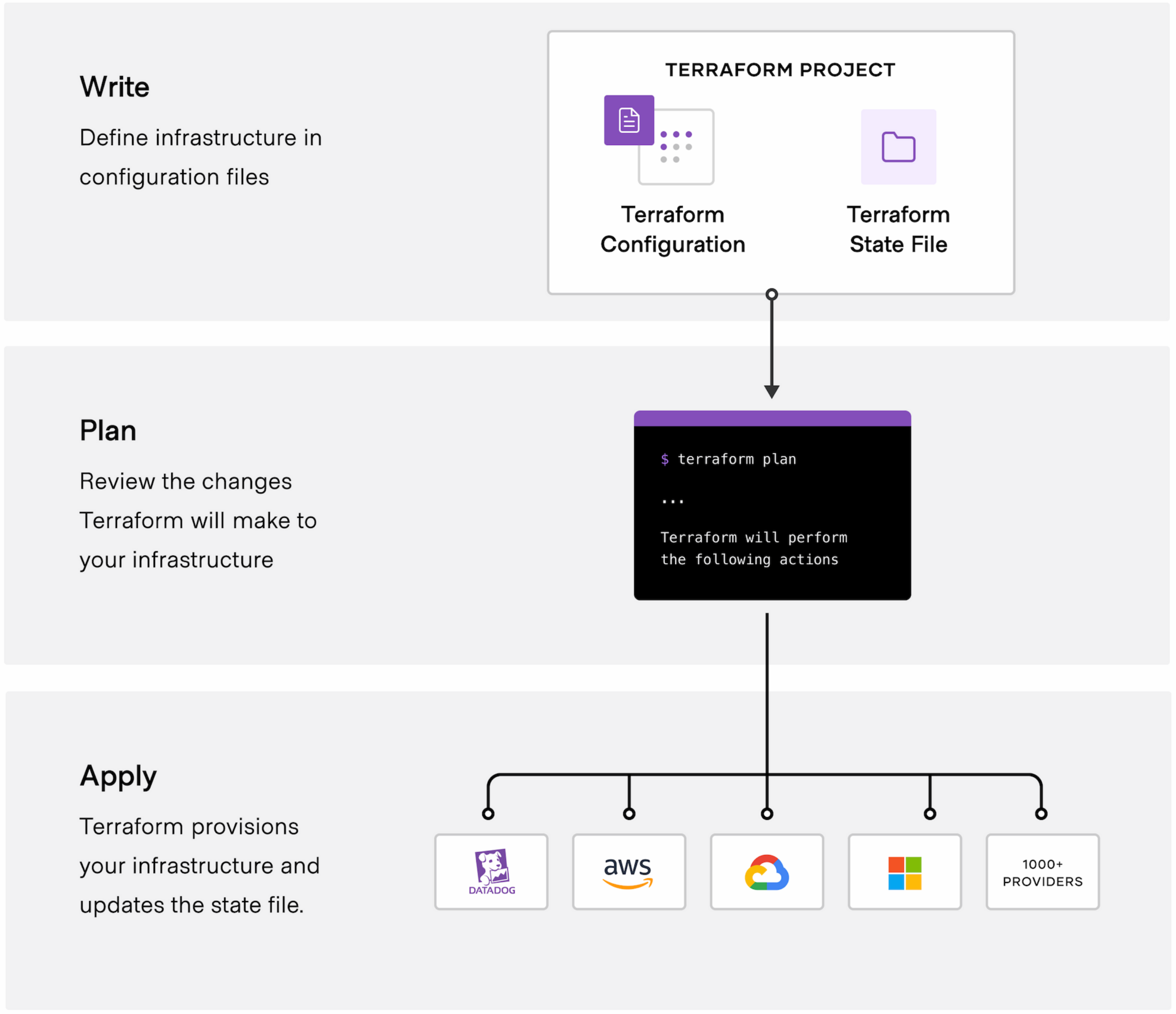
How Terraform work ?

HashiCorp Terraform

Terraform components

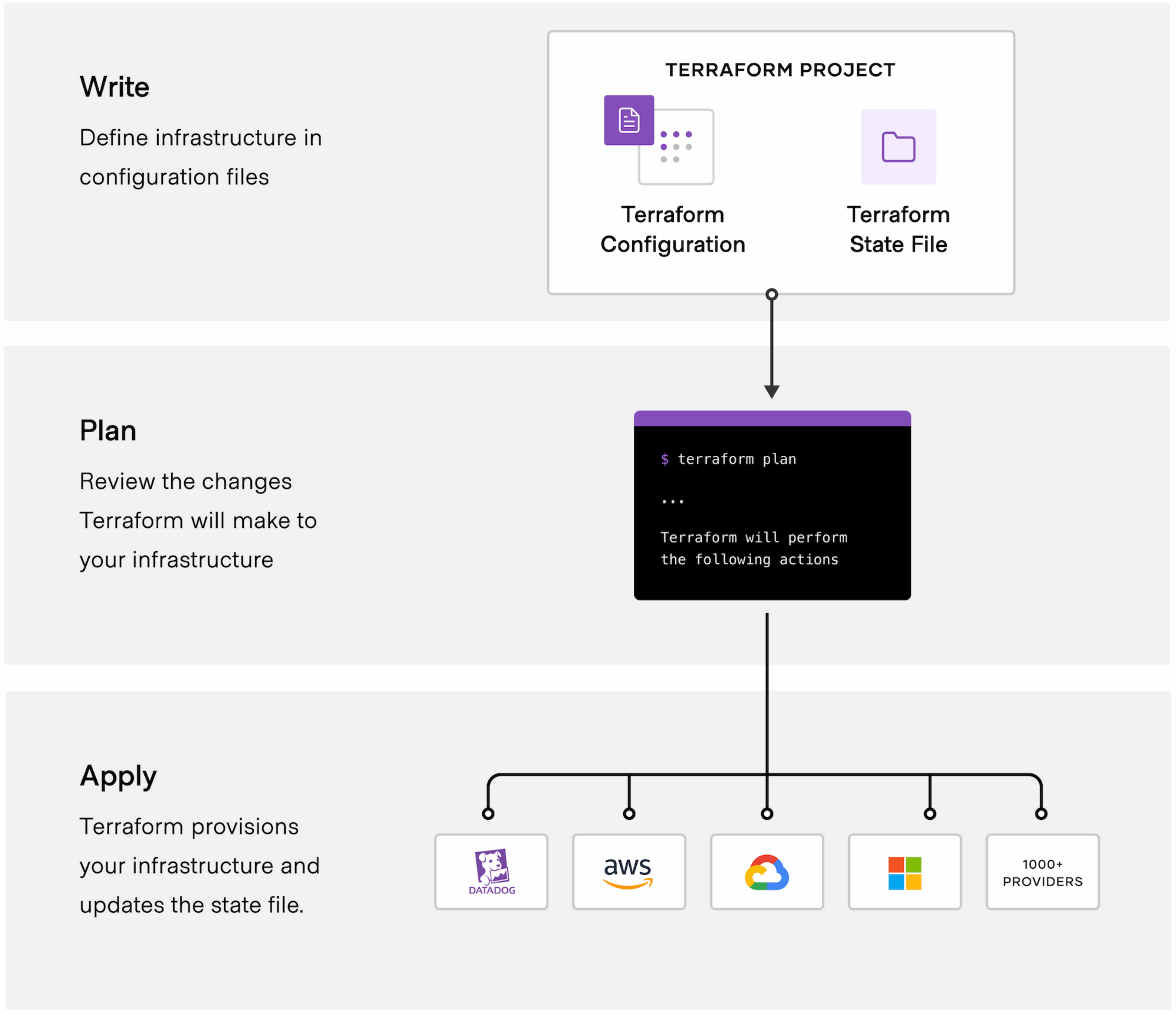


Terraform workflow


Write
Define infrastructure in configuration files
TERRAFORM PROJECT
Terraform Configuration
Terraform State File
Plan
Review the changes Terraform will make to your infrastructure
Apply
Terraform provisions your infrastructure and updates the state file.
DATADOG
aws
1000+ PROVIDERS

```
$ terraform plan

...

Terraform will perform
the following actions
```

Demo



Write

Define infrastructure in configuration files

Plan

Review the changes Terraform will make to your infrastructure

Apply

Terraform provisions your infrastructure and updates the state file.

```
$ terraform plan
...
Terraform will perform
the following actions
```

Other topics

Enterprise scale landing zone

Infracost

Terragrunt

Terratest

How it works

- Variables
  - Values can be supplied as a .tfvars file containing simple key/value pairs, env variables, or command parameters.
- Functions
  - String and math (all the usual)
  - Count – simple method for deploying multiple resources
  - Conditional "${var.env == "production" ? var.prod_subnet : var.dev_subnet}"
- Provisioners
  - local-exec, remote-exec, file
- Modules
  - Reusable template code
  - Container of resources that are used together

```
module "vaultstorage" {
  source = "./modules/storage/account"

  name                = "${var.vault_storage_account_name}"
  resource_group_name = "${azurerm_resource_group.storage_rg.name}"
  location            = "${azurerm_resource_group.storage_rg.location}"

  tags {
    Application = "Vault"
    Environment = "SS-Prod"
    Category    = "Storage Account"
  }
}
```

How it works (Continued)

- State
  - Responsible for mapping *"azure_virtual_machine" "vm"* to *"/subscriptions/dcf628c7-fc9d-4e40-af2c-5c963345a237/resourceGroups/BDIterraformdemo/providers/Microsoft.Compute/virtualMachines/BDIvm-vm"*
  - Tracks dependencies between resources
    - Knows that if the VM is deleted, to also delete the Disk(s)
  - Provides the ability to pass in previous deployments as parameters

How to use it

- Terraform ships as a standalone executable
- Requires Azure CLI
- Code written in .tf files
- Perform deployments with the following commands:
  - `terraform init` – downloads referenced providers locally
  - `terraform plan` – displays a list of all resources that will be added or removed
  - `terraform apply` – deploys resources
  - `terraform destroy` – deletes resources

ARM Template/Terraform Template

```
{
  "$schema": "http://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
  "contentVersion": "1.0.0.0",
  "parameters": {
      "location": {
          "type": "string"
      },
      "accountType": {
          "type": "string"
      },
      "kind": {
          "type": "string"
     }
  },
  "variables": {
      "storageAccountName": "[concat(uniquestring(resourceGroup().id), 'standardsa')]"
  },
  "resources": [
      {
          "name": "[variables('storageAccountName')]",
          "type": "Microsoft.Storage/storageAccounts",
          "apiVersion": "2018-07-01",
          "location": "[parameters('location')]",
          "properties": {
              "accessTier": "[parameters('accessTier')]",
              "supportsHttpsTrafficOnly": "[parameters('supportsHttpsTrafficOnly')]"
          },
          "dependsOn": [],
          "sku": {
              "name": "[parameters('accountType')]"
          },
          "kind": "[parameters('kind')]"
      }
  ],
  "outputs": {}
}
```

```
variable “location" {
  type     = “string”
}

variable “accountType" {
  type     = “string”
}

variable “tier" {
  type     = “string”
}

resource "random_string" "rnd"{
  length = 8
  special = false
  upper = false
  lower = true
}

resource "azurerm_resource_group" "testrg" {
  name     = "resourceGroupName"
  location = “${var.location}”
}

resource "azurerm_storage_account" "testsa" {
  name                      = “${randomstring.rnd.result}storacc"
  resource_group_name       = "${azurerm_resource_group.testrg.name}"
  location                  = “${var.location}”
  account_tier              = “$var.tier"
  account_replication_type  = “${var.accountType}“
  enable_https_traffic_only = true

}
```

Variables

Dependencies

Functions

Resources

Microsoft

Language elements and details

HashiCorp Configuration Language (HCL)

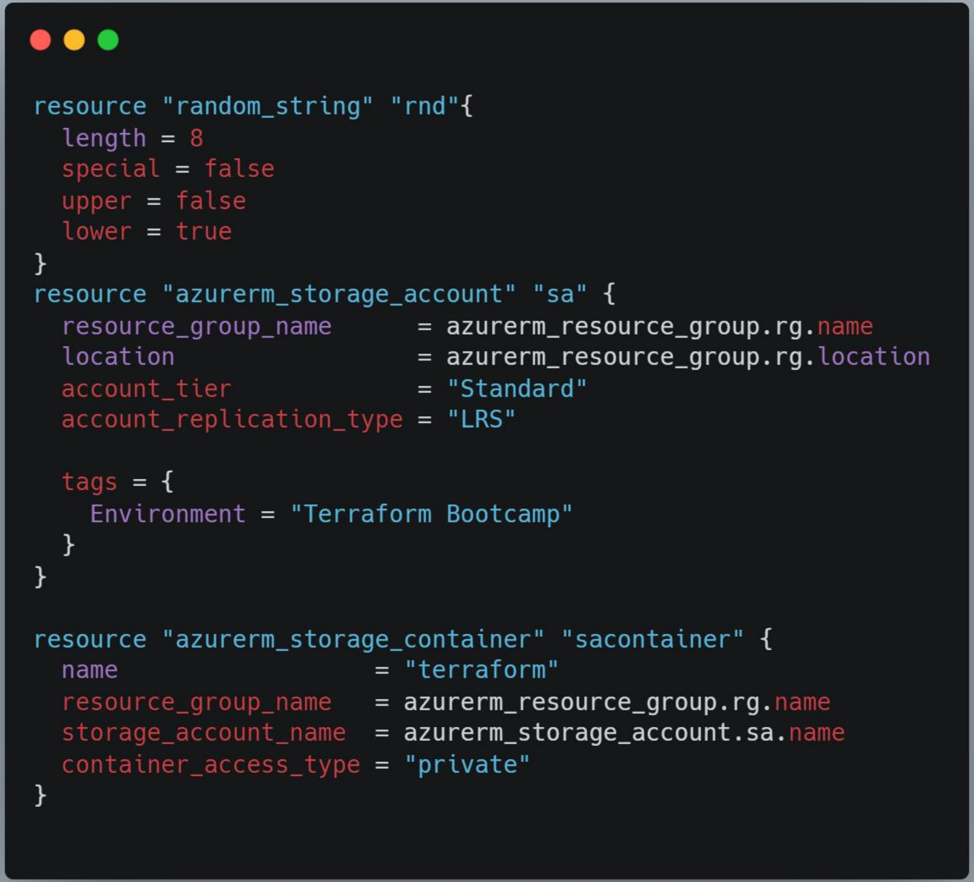
HCL is the simple, YAML-like language used in Terraform files.

Blocks of code containing **key/value pairs** are combined to create a deployment. The key/value pairs define the configuration of the resource to be created.

.

Interpolation Syntax has been simplified in the most recent version of terraform. Resources, functions and calculations will all be calculated automatically. Or, if you need to combine strings with calculation, all code contained within ***${}*** is evaluated at time of deployment to determine configuration values.

Comments in HCL can be specified using **#**, ***//*** or ***/* */***



```
resource "random_string" "rnd"{
  length = 8
  special = false
  upper = false
  lower = true
}
resource "azurerm_storage_account" "sa" {
  resource_group_name      = azurerm_resource_group.rg.name
  location                 = azurerm_resource_group.rg.location
  account_tier             = "Standard"
  account_replication_type = "LRS"

  tags = {
    Environment = "Terraform Bootcamp"
  }
}

resource "azurerm_storage_container" "sacontainer" {
  name                  = "terraform"
  resource_group_name   = azurerm_resource_group.rg.name
  storage_account_name  = azurerm_storage_account.sa.name
  container_access_type = "private"
}
```

File Structure

Terraform **configuration files** are named with the **.tf** extension

A deployment can be created using a **single .tf file** containing all of the code for a deployment; or can be split into **multiple .tf files** to facilitate code management.

A common file structure to use is:

- variables.tf
- main.tf
- output.tf
- (backend.tf, locals.tf, provider.tf, etc.)

Files can use **any name**, but **must include** the **.tf** extension, and all files for a given deployment must be stored in the **same directory**.

The other important configuration file is the Terraform variables file which uses the **.tfvars** extension. This provides a mechanism to store input variables separately from the configuration. Variables will be covered in more detail in a subsequent slide.

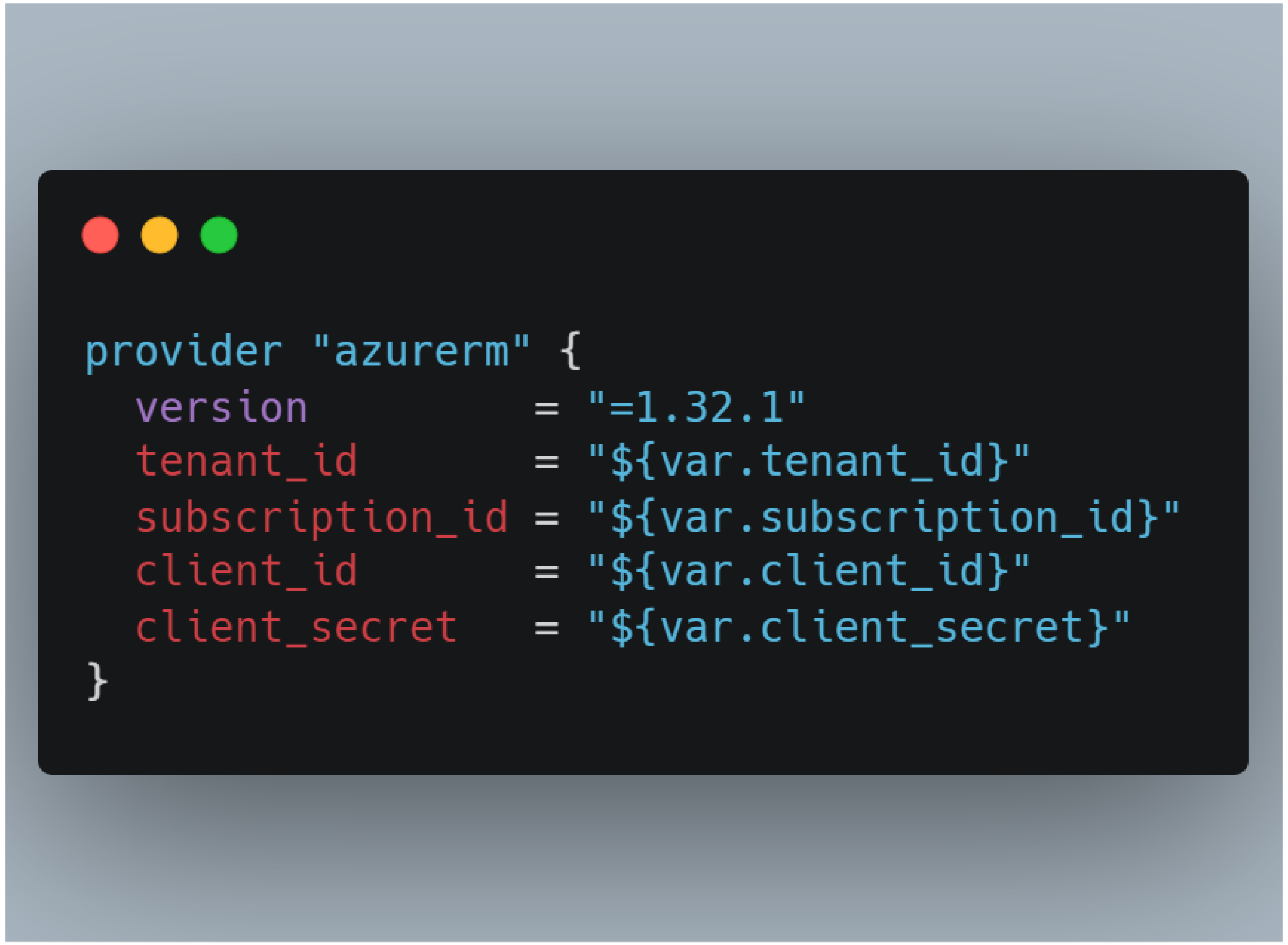
Provider

The ***provider*** block is **optional** in simple scenarios. It can be used in a Terraform configuration to control provider version, and in the case of the **azurerm** provider, to specify subscription ID, and service principal details.

Using the ***alias*** **argument**, **multiple provider** blocks can be specified to allow deployments to run against multiple tenants, or multiple subscriptions.

When using a provider alias, the ***provider*** **argument** can be used within resource blocks to control which subscription a resources will be deployed to.

If using provider blocks with **credentials**, do not check them into source control. Create a provider.tf file and add that to your .**gitignore** file.



```
provider "azurerm" {
  version         = "=1.32.1"
  tenant_id       = "${var.tenant_id}"
  subscription_id = "${var.subscription_id}"
  client_id       = "${var.client_id}"
  client_secret   = "${var.client_secret}"
}
```

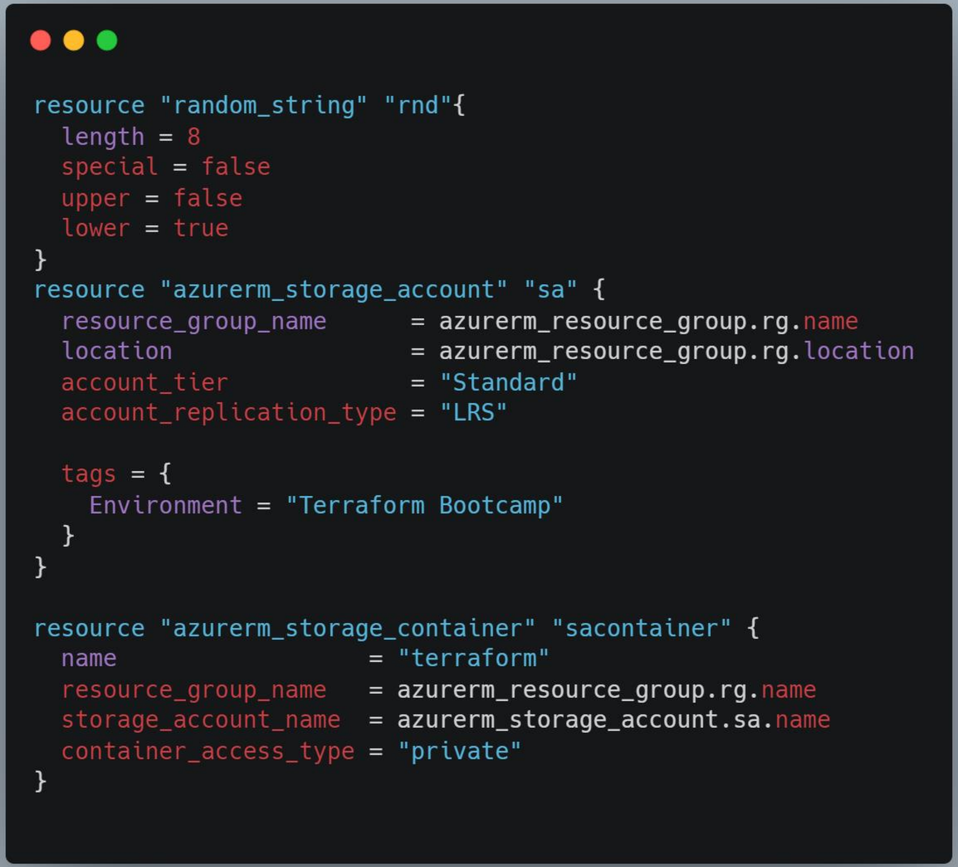
Resources

resource blocks define the infrastructure components to be deployed by Terraform. They are where the magic happens!

When defining a resource, the **resource type** must be specified (e.g. “azurerm_storage_account”) followed by an arbitrary **resource name** (e.g. “teststorageaccount”). The resource name is used to reference the resource throughout the configuration.

The **resource block contains** the **configuration arguments** for the resource such as name, location, and service specific arguments such as Storage Replication Type. Each resource will have different arguments – refer to the documentation for details.

ACTION: Review the azurerm_storage_account resource documentation to see the arguments that can be used.



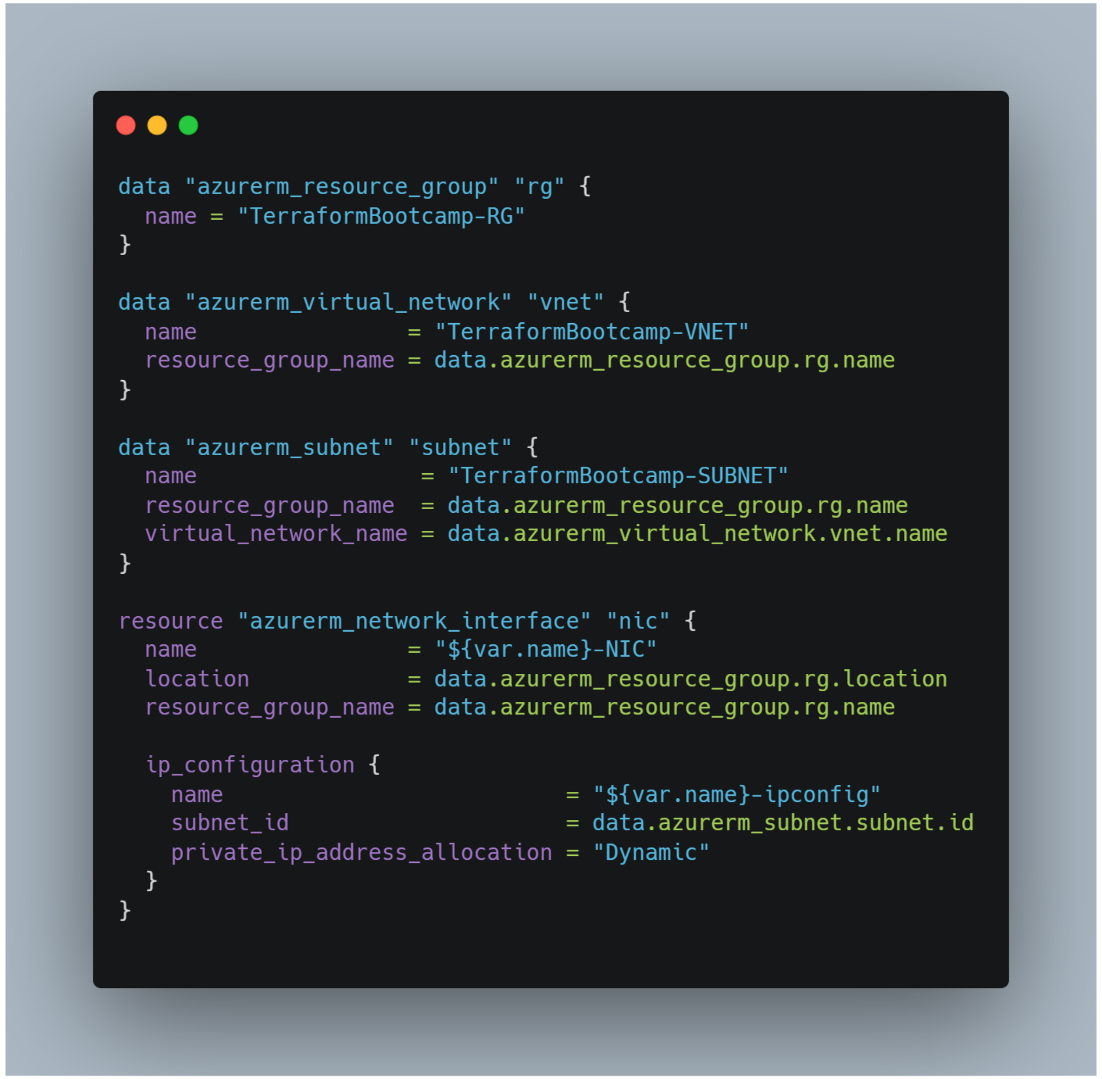
```
resource "random_string" "rnd"{
  length = 8
  special = false
  upper = false
  lower = true
}
resource "azurerm_storage_account" "sa" {
  resource_group_name      = azurerm_resource_group.rg.name
  location                 = azurerm_resource_group.rg.location
  account_tier             = "Standard"
  account_replication_type = "LRS"

  tags = {
    Environment = "Terraform Bootcamp"
  }
}

resource "azurerm_storage_container" "sacontainer" {
  name                  = "terraform"
  resource_group_name   = azurerm_resource_group.rg.name
  storage_account_name  = azurerm_storage_account.sa.name
  container_access_type = "private"
}
```

Data Sources

When referencing the azurerm provider documentation, note that many resources are listed twice – once under the heading **Data Sources** and once under the resource category (e.g. Storage Resources). Data sources use the **data** block to **reference existing resources** and have fewer parameters.

The attributes obtained using a data block can be referenced by other resources
(eg. `data.azurerm_storage_account.teststorageaccount`)

ACTION: Review the azurerm_storage_account data source documentation to see the attributes that that can be obtained.



```
data "azurerm_resource_group" "rg" {
  name = "TerraformBootcamp-RG"
}

data "azurerm_virtual_network" "vnet" {
  name                = "TerraformBootcamp-VNET"
  resource_group_name = data.azurerm_resource_group.rg.name
}

data "azurerm_subnet" "subnet" {
  name                 = "TerraformBootcamp-SUBNET"
  resource_group_name  = data.azurerm_resource_group.rg.name
  virtual_network_name = data.azurerm_virtual_network.vnet.name
}

resource "azurerm_network_interface" "nic" {
  name                = "${var.name}-NIC"
  location            = data.azurerm_resource_group.rg.location
  resource_group_name = data.azurerm_resource_group.rg.name

  ip_configuration {
    name                          = "${var.name}-ipconfig"
    subnet_id                     = data.azurerm_subnet.subnet.id
    private_ip_address_allocation = "Dynamic"
  }
}
```

HCL - Variables

To increase reusability, configuration parameters in Terraform can be set using the **variable** block.

Each variable is represented in its own block where attributes such as ***description*** and ***type*** can be set, as well as ***default*** values.

The variable types are a combination of a type keyword and type constructors.

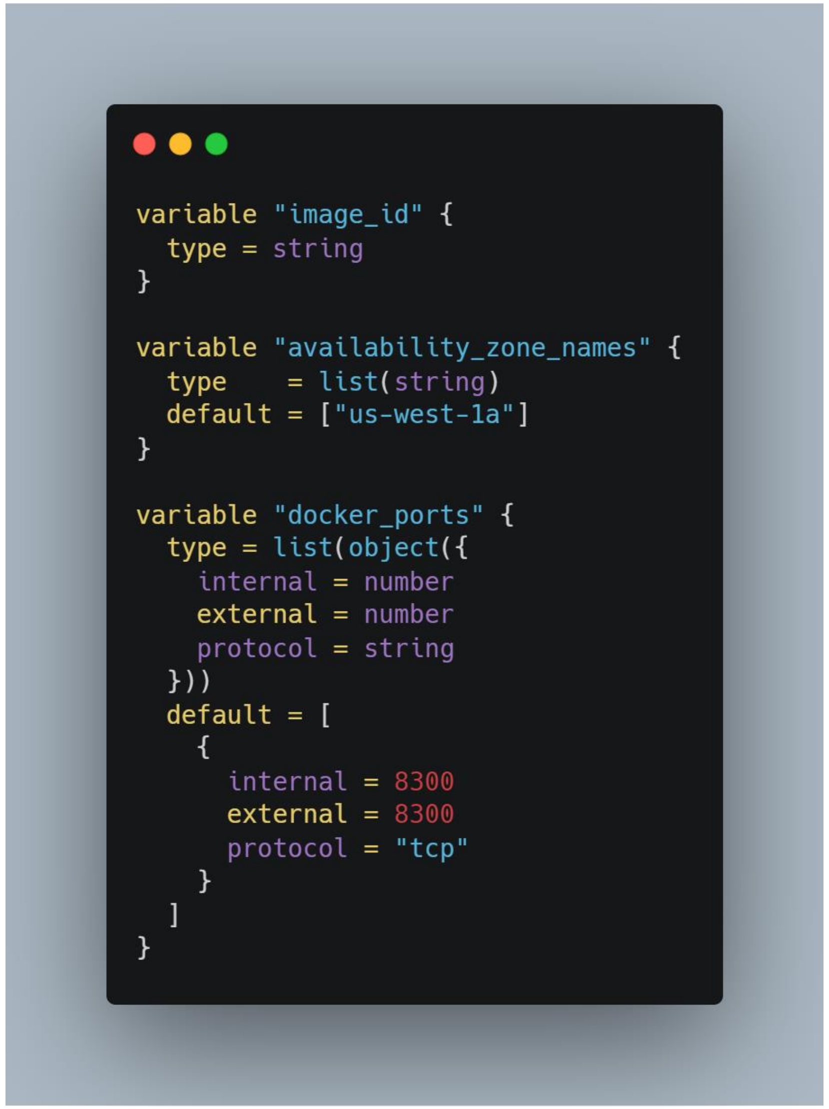
Keywords:

- string
- number
- bool

Constructor

- list
- set
- map
- object
- tuple

Variables can be referenced within a deployment using by specifying **var** followed by the variable name. ***var.example***



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

variable "availability_zone_names" {
  type    = list(string)
  default = ["us-west-1a"]
}

variable "docker_ports" {
  type = list(object({
    internal = number
    external = number
    protocol = string
  }))
  default = [
    {
      internal = 8300
      external = 8300
      protocol = "tcp"
    }
  ]
}
```

Variables

As mentioned previously, it is common to declare variables in a separate *variables.tf* file. Once declared, there are several methods for assigning values to the variables:

- A **.tfvars** file can be created with simple key value pairs representing the variable values (Note: Any .tfvars file containing sensitive information should not be stored in source control). The tfvars file is then called using the ***–var-file*** command line parameter.
- The **-var** parameter can be used when executing terraform plan, apply and destroy commands. Multiple –***var*** parameters can be supplied to each command.
- Creating an **Environment variable** prefixed with ***TF_VARS_<variable name>*** can also be used to set values. These are automatically referenced when executing terraform commands.

Another benefit of variables is they avoid the need to store sensitive information in template files. For example in the **provider** block covered in an earlier slide, the subscription, tenant and service principal details are replaced with variables.

Use tfvars file with string, list and map variable types:

```
> terraform plan -var-file=example.tfvars
```

Supplying variable values at command line string, list, map :

```
> terraform plan -var storageaccountname="storage123" –
var regions='["eastasia","southeastasia"]' -var
tags='{environment = "Production", owner = "Finance"}'
```

Creating environment variables (Bash example) string, list, map:

```
> TF_VAR_storageaccountname="storage123"
> TF_VAR_regions='["eastasia","southeastasia"]'
> TF_VAR_tags='{environment = "Production", owner =
"Finance"}'
> terraform plan
```

Interpolation and Functions

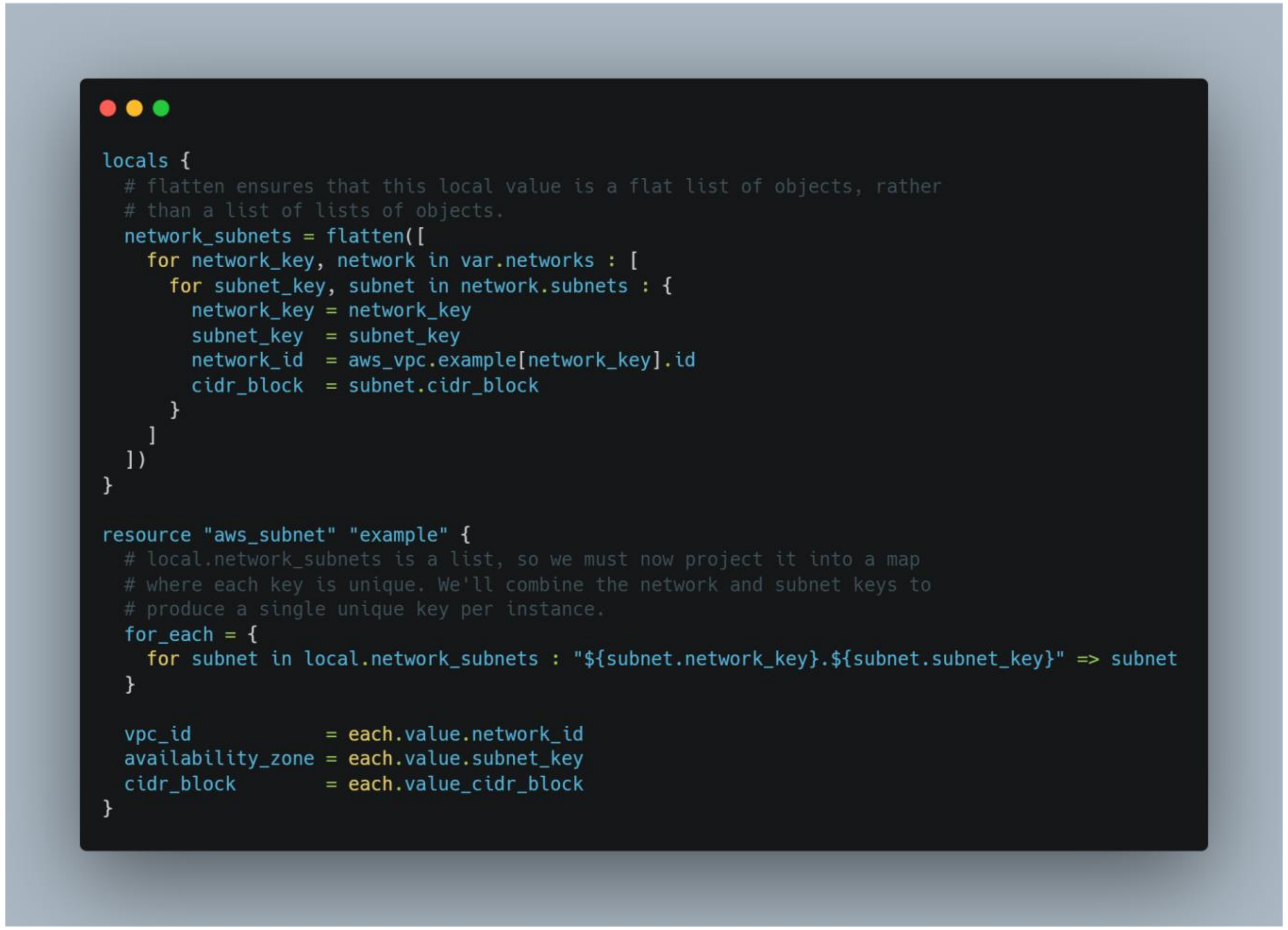
We have mentioned **interpolation syntax** a few times throughout this presentation – it is a simple, yet powerful mechanism for creating dynamic deployments by looking up and calculating attribute values during execution.

It is used to reference **variables** and **resource attributes**, perform **simple math**, call **functions**, and determine values based on **conditionals**.

Attributes from other resources can be referenced using the syntax ***<resource type>.<resource name>.<attribute>*** *eg* ***azurerm_storage_account.storageaccount.name*** .

Functions provide more complex interpolation capabilities. They are called using the syntax **functionname(arg1, arg2, ...)** Here are some commonly used functions.

Conditionals are used to determine a value based on a true/false condition using the format: *condition ? true_val : false_val*



```
locals {
  # flatten ensures that this local value is a flat list of objects, rather
  # than a list of lists of objects.
  network_subnets = flatten([
    for network_key, network in var.networks : [
      for subnet_key, subnet in network.subnets : {
        network_key = network_key
        subnet_key  = subnet_key
        network_id  = aws_vpc.example[network_key].id
        cidr_block  = subnet.cidr_block
      }
    ]
  ])
}

resource "aws_subnet" "example" {
  # local.network_subnets is a list, so we must now project it into a map
  # where each key is unique. We'll combine the network and subnet keys to
  # produce a single unique key per instance.
  for_each = {
    for subnet in local.network_subnets : "${subnet.network_key}.${subnet.subnet_key}" => subnet
  }

  vpc_id            = each.value.network_id
  availability_zone = each.value.subnet_key
  cidr_block        = each.value_cidr_block
}
```

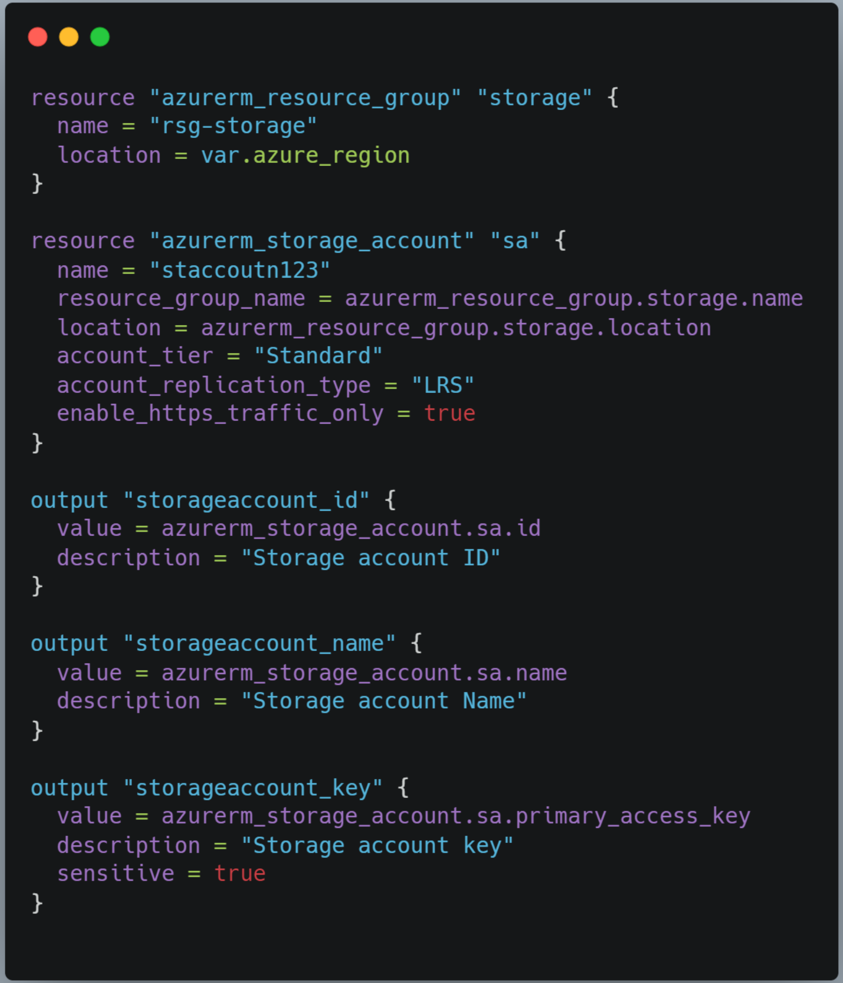
Outputs

Attribute values generated during a terraform deployment can be defined as an **output**. Outputs are displayed on screen when executing terraform on the command line, and stored in the **state** file.

This is useful for displaying dynamically generates attributes, such as VM names, IP addresses or storage account keys.

An **output** block must be defined for each value you wish to output. The output can be assigned an arbitrary name, and uses the ***value*** key to set the output value (Interpolation syntax can be used). You can also add a ***description***, and mark the value as ***sensitive*** to prevent displaying the output on screen.

The attributes available to output vary between resource types. The documentation for each resource has an **attribute reference** list.



```
resource "azurerm_resource_group" "storage" {
  name = "rsg-storage"
  location = var.azure_region
}

resource "azurerm_storage_account" "sa" {
  name = "staccoutn123"
  resource_group_name = azurerm_resource_group.storage.name
  location = azurerm_resource_group.storage.location
  account_tier = "Standard"
  account_replication_type = "LRS"
  enable_https_traffic_only = true
}

output "storageaccount_id" {
  value = azurerm_storage_account.sa.id
  description = "Storage account ID"
}

output "storageaccount_name" {
  value = azurerm_storage_account.sa.name
  description = "Storage account Name"
}

output "storageaccount_key" {
  value = azurerm_storage_account.sa.primary_access_key
  description = "Storage account key"
  sensitive = true
}
```

Microsoft

Using Terraform

State

Terraform stores the **state** of your infrastructure in a file called ***terraform.tfstate.*** The state file is created and/or referenced when planning and making changes to your configuration.

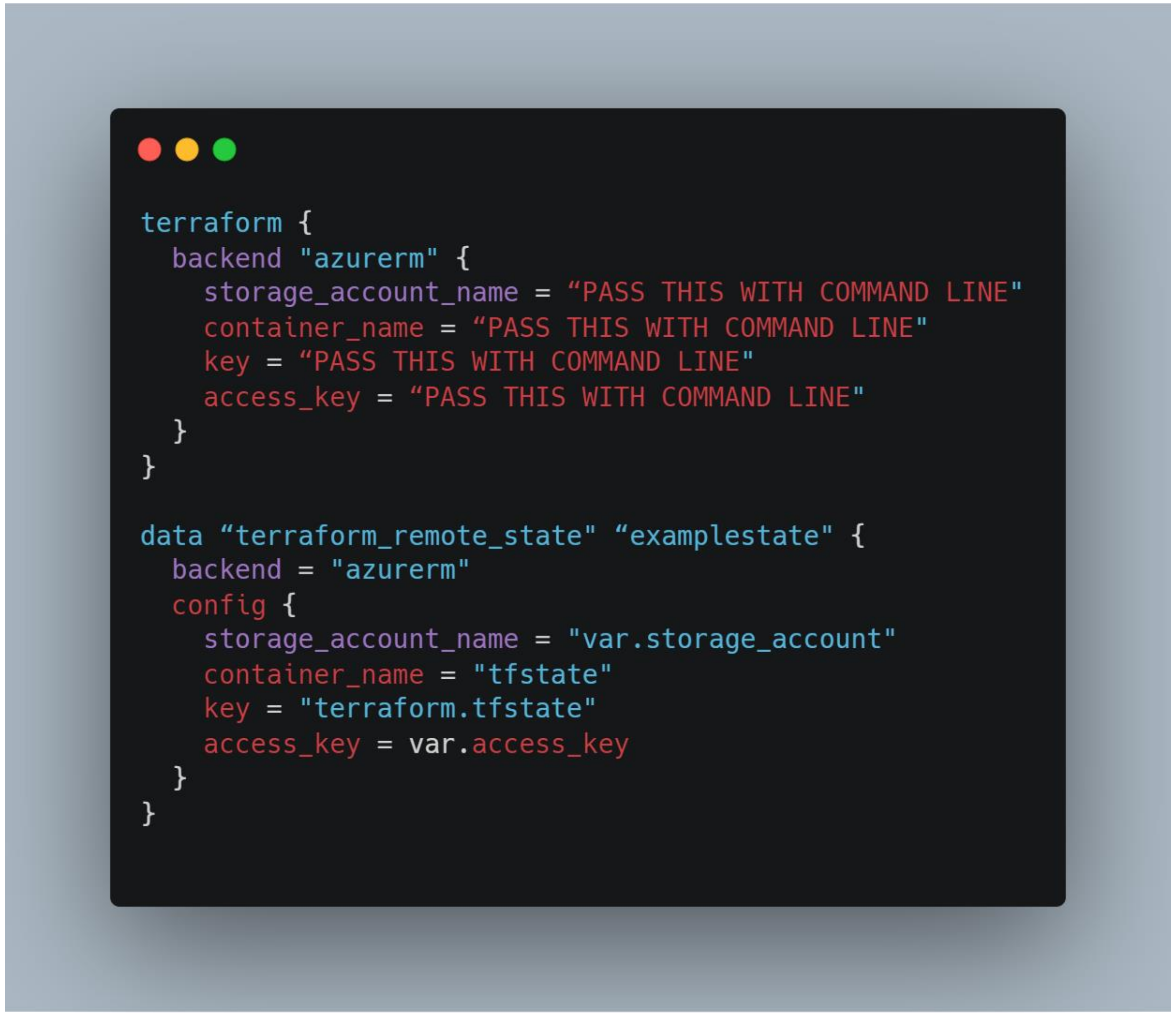
The state file is **critical for mapping your configuration files to real world resources** along with any meta data such as resource dependencies. It also improves performance of deployments.

By default terraform will use a ***local*** state file. The file is automatically created as ***terraform.tfstate*** in the directory containing your configuration. This file should not be edited directly, it is updated each time you run the **apply** and **destroy** commands.

The state file must be protected from loss or corruption as it contains the Terraform to real-world mapping. The **import** command can be used **to recover a lost or corrupted state file**, however **full recovery is not guaranteed**, and can be quite complex – so make sure you protect your state!

The state file must also be protected from unauthorised access as it **can contain sensitive data**.

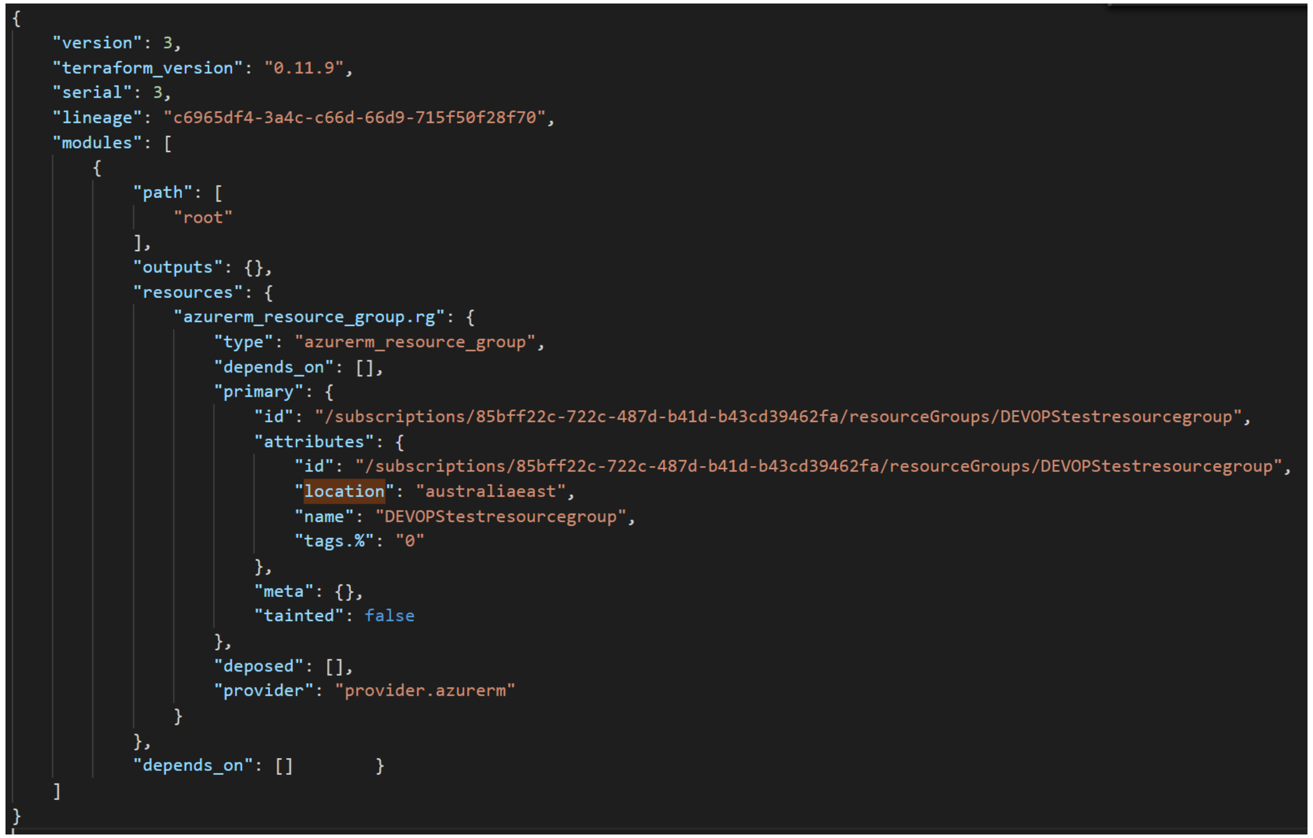
Further Information:
Terraform Remote State



```
terraform {
  backend "azurerm" {
    storage_account_name = “PASS THIS WITH COMMAND LINE"
    container_name = “PASS THIS WITH COMMAND LINE"
    key = “PASS THIS WITH COMMAND LINE"
    access_key = “PASS THIS WITH COMMAND LINE"
  }
}

data “terraform_remote_state" “examplestate" {
  backend = "azurerm"
  config {
    storage_account_name = "var.storage_account"
    container_name = "tfstate"
    key = "terraform.tfstate"
    access_key = var.access_key
  }
}
```

State

*Example State file – the **only resource** in this deployment **is a single resource group**. This is a very simple example – state files get very big, very quickly as you deploy more resources* updated each time you run the **apply** and **destroy** commands.



```
{
    "version": 3,
    "terraform_version": "0.11.9",
    "serial": 3,
    "lineage": "c6965df4-3a4c-c66d-66d9-715f50f28f70",
    "modules": [
        {
            "path": [
                "root"
            ],
            "outputs": {},
            "resources": {
                "azurerm_resource_group.rg": {
                    "type": "azurerm_resource_group",
                    "depends_on": [],
                    "primary": {
                        "id": "/subscriptions/85bff22c-722c-487d-b41d-b43cd39462fa/resourceGroups/DEVOPStestresourcegroup",
                        "attributes": {
                            "id": "/subscriptions/85bff22c-722c-487d-b41d-b43cd39462fa/resourceGroups/DEVOPStestresourcegroup",
                            "location": "australiaeast",
                            "name": "DEVOPStestresourcegroup",
                            "tags.%": "0"
                        },
                        "meta": {},
                        "tainted": false
                    },
                    "deposed": [],
                    "provider": "provider.azurerm"
                }
            },
            "depends_on": []        }
    ]
}
```

The State file contains a **mapping of your terraform resource** e.g. ("azure_virtual_machine" "vm") to **the real world resource id** (e.g. "/subscriptions/edf628c7-fc1d-4e40-af2c-5c343345a124/resourceGroups/terraformdemo/providers/Microsoft.Compute/virtualMachines/vm1")

Further Information:
[The Purpose of Terraform State]
[Terraform Import Command]

Remote State

Using a local state file works for individuals working with Terraform, however if multiple people maintain the Terraform configuration configuring **remote state** storage becomes essential. To do this you must add a **terraform backend** block to your configuration.

A **backend** can refer to a shared storage location, such as an **Azure Storage Account**. Storage Accounts also **support file locking** which **prevents** multiple people from **simultaneously executing a Terraform deployment**. Before each deployment, Terraform must be able to obtain a lock on the state file. Once the deployment is complete, the lock is released.

A properly configured **storage account** can provide **security, version control** and **disaster recovery** capabilities for your state files.

State files can be **referenced across deployments** using a **data** block (*see an exmple on the next slide*). This is useful for linking multiple Terraform deployments. Rather than having one monolithic deployment you can divide deployments based on (for example) security requirements, or a tiered deployment model (e.g. Hub, Spoke). Only **outputs** stored in a state file **are accessible using a data block**.

Remote state is also essential when performing Terraform deployments using DevOps pipelines and Agent pools.

Further Information:
[Terraform Remote State]

Environment Setup

To use Terraform, you must take care of the following prerequisites:

- Install Azure CLI - https://docs.microsoft.com/en-us/cli/azure/install-azure-cli?view=azure-cli-latest
- Download Terraform standalone executable - https://www.terraform.io/downloads.html
  - The download contains a zip file with a single executable file "terraform.exe"
  - Copy it to a location in your systems **PATH**, or store it in another location and update your PATH environment variable.
- To take part in the training exercises you will also need an Azure subscription!

Alternatively Terraform is installed by default in Azure Cloud Shell – all prerequisites taken care of!

terraform init

Terraform commands are generally executed in the directory that contains the ***.tf** files to be deployed.

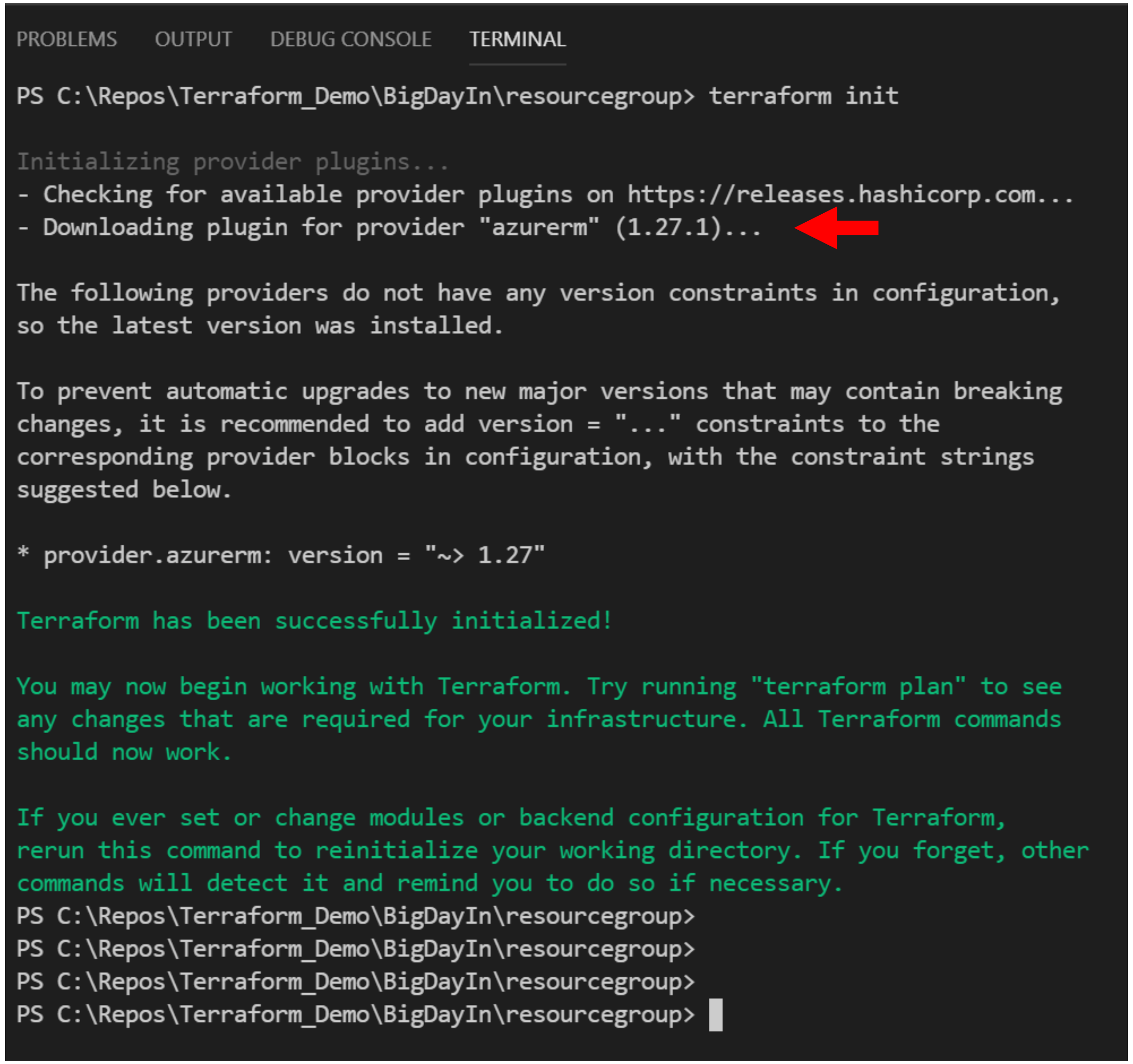
The ***init*** command must be executed first to **download providers and modules** for the first time, and to **update** providers and modules. It determines which providers to download based on the .tf files in the current directory.

Provider files are stored within the current directory in a hidden directory "***.terraform***"

Before running ***terraform init***, connect to Azure using az cli to obtain an authorization token, and select the subscription you wish to deploy to.

```
az login
az account set –subscription <subscriptionid>
```

Further Information:
Terraform Init Command
Azure CLI Subscription Documentation



```
PS C:\Repos\Terraform_Demo\BigDayIn\resourcegroup> terraform init

Initializing provider plugins...
- Checking for available provider plugins on https://releases.hashicorp.com...
- Downloading plugin for provider "azurerm" (1.27.1)...

The following providers do not have any version constraints in configuration,
so the latest version was installed.

To prevent automatic upgrades to new major versions that may contain breaking
changes, it is recommended to add version = "..." constraints to the
corresponding provider blocks in configuration, with the constraint strings
suggested below.

* provider.azurerm: version = "~> 1.27"

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
PS C:\Repos\Terraform_Demo\BigDayIn\resourcegroup>
PS C:\Repos\Terraform_Demo\BigDayIn\resourcegroup>
PS C:\Repos\Terraform_Demo\BigDayIn\resourcegroup>
PS C:\Repos\Terraform_Demo\BigDayIn\resourcegroup>
```

terraform plan

Further Information:
Terraform Plan Command

Next, execute the **plan** command to provide a preview of the actions that terraform will take. This provides an opportunity to sanity check the deployment, paying attention to any resource changes or removals indicated. No actual changes will be made.

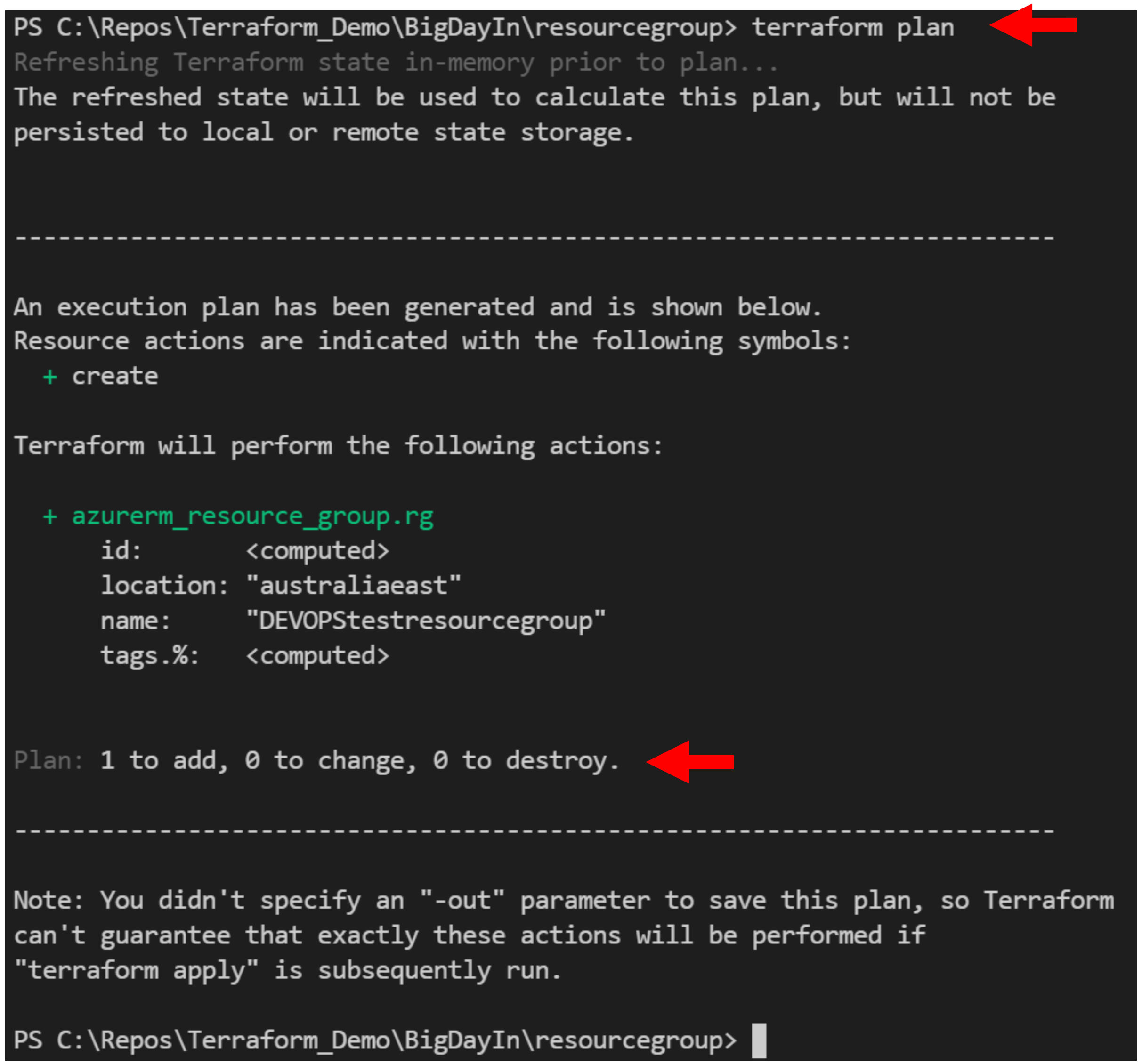
Resource **additions** will be indicated by a + symbol, and highlighted in green

Resource **modifications** will be indicated by a ~ symbol and highlighted in orange

Resource **deletions** will be indicated by a – symbol and highlighted in red

(Note: Colours only shown if supported by terminal)

Where required, specify the ***–var*** and ***–var-file*** parameters to pass in variable values to the **plan** command.



```
PS C:\Repos\Terraform_Demo\BigDayIn\resourcegroup> terraform plan
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but will not be
persisted to local or remote state storage.


------------------------------------------------------------------------


An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  + azurerm_resource_group.rg
      id:        <computed>
      location:  "australiaeast"
      name:      "DEVOPStestresourcegroup"
      tags.%:    <computed>


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

------------------------------------------------------------------------


Note: You didn't specify an "-out" parameter to save this plan, so Terraform
can't guarantee that exactly these actions will be performed if
"terraform apply" is subsequently run.

PS C:\Repos\Terraform_Demo\BigDayIn\resourcegroup>
```

terraform apply

Further Information:
Terraform Apply Command

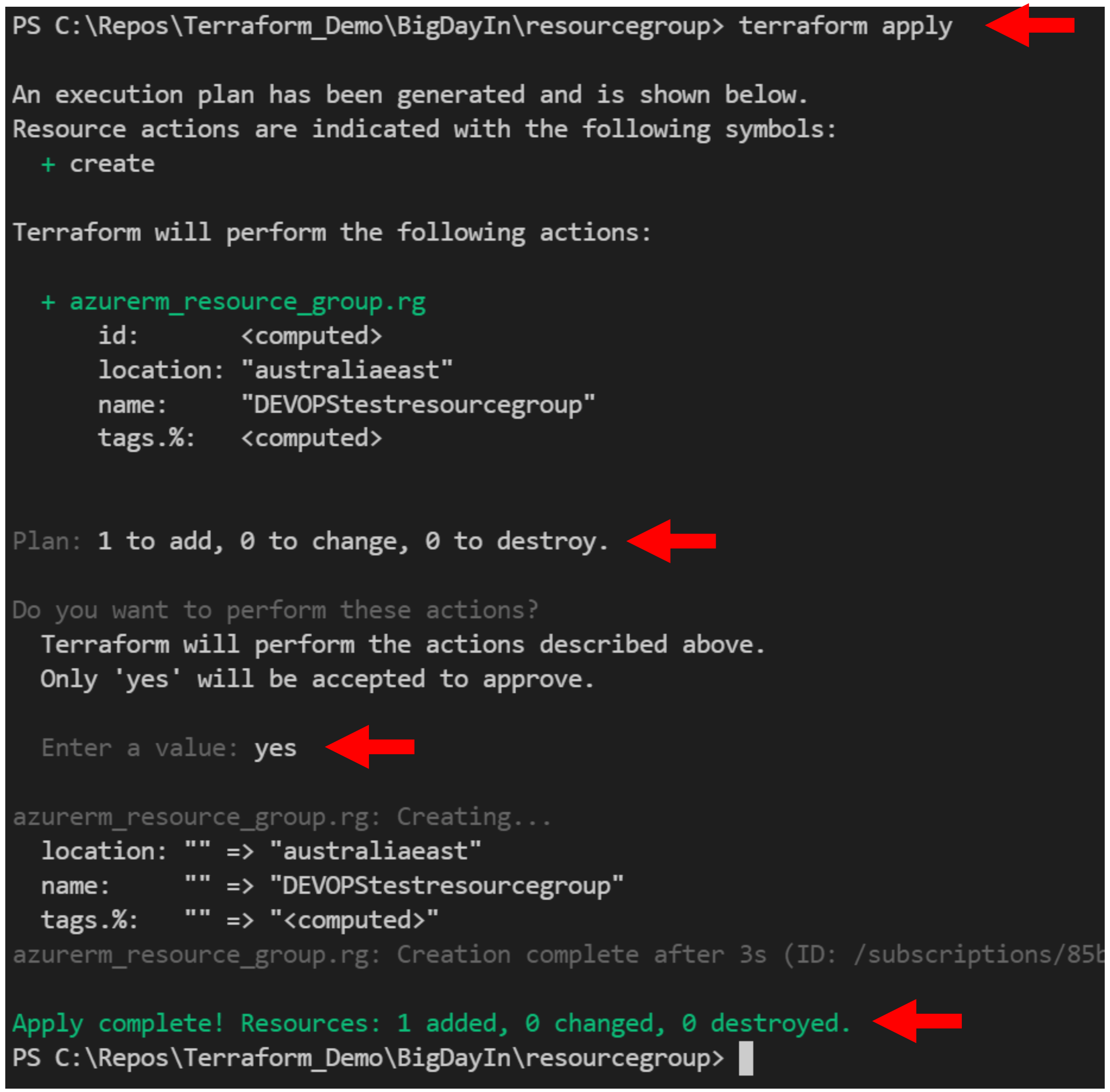
The **apply** command is used to execute the terraform deployment and add, change and delete resources. When executed, you are presented with a similar output to the **plan** command and then prompted to confirm before continuing.

To facilitate more automated deployments the ***-auto-approve*** parameter can be specified to skip the confirmation step.

Where required, specify the ***–var*** and ***–var-file*** parameters to pass in variable values to the **apply** command.

This command can take some time to run for large deployments. Continuous output will be provided to assist in monitoring progress.

Updating deployments: Terraform is not simply a deployment technology. Once a deployment has been completed it can be maintained by updating the .tf files for complete lifecycle management using infrastructure-as-code.



```
PS C:\Repos\Terraform_Demo\BigDayIn\resourcegroup> terraform apply

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  + azurerm_resource_group.rg
      id:        <computed>
      location:  "australiaeast"
      name:      "DEVOPStestresourcegroup"
      tags.%:    <computed>


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

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes

azurerm_resource_group.rg: Creating...
  location: "" => "australiaeast"
  name:     "" => "DEVOPStestresourcegroup"
  tags.%:   "" => "<computed>"
azurerm_resource_group.rg: Creation complete after 3s (ID: /subscriptions/85

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
PS C:\Repos\Terraform_Demo\BigDayIn\resourcegroup>
```

terraform destroy

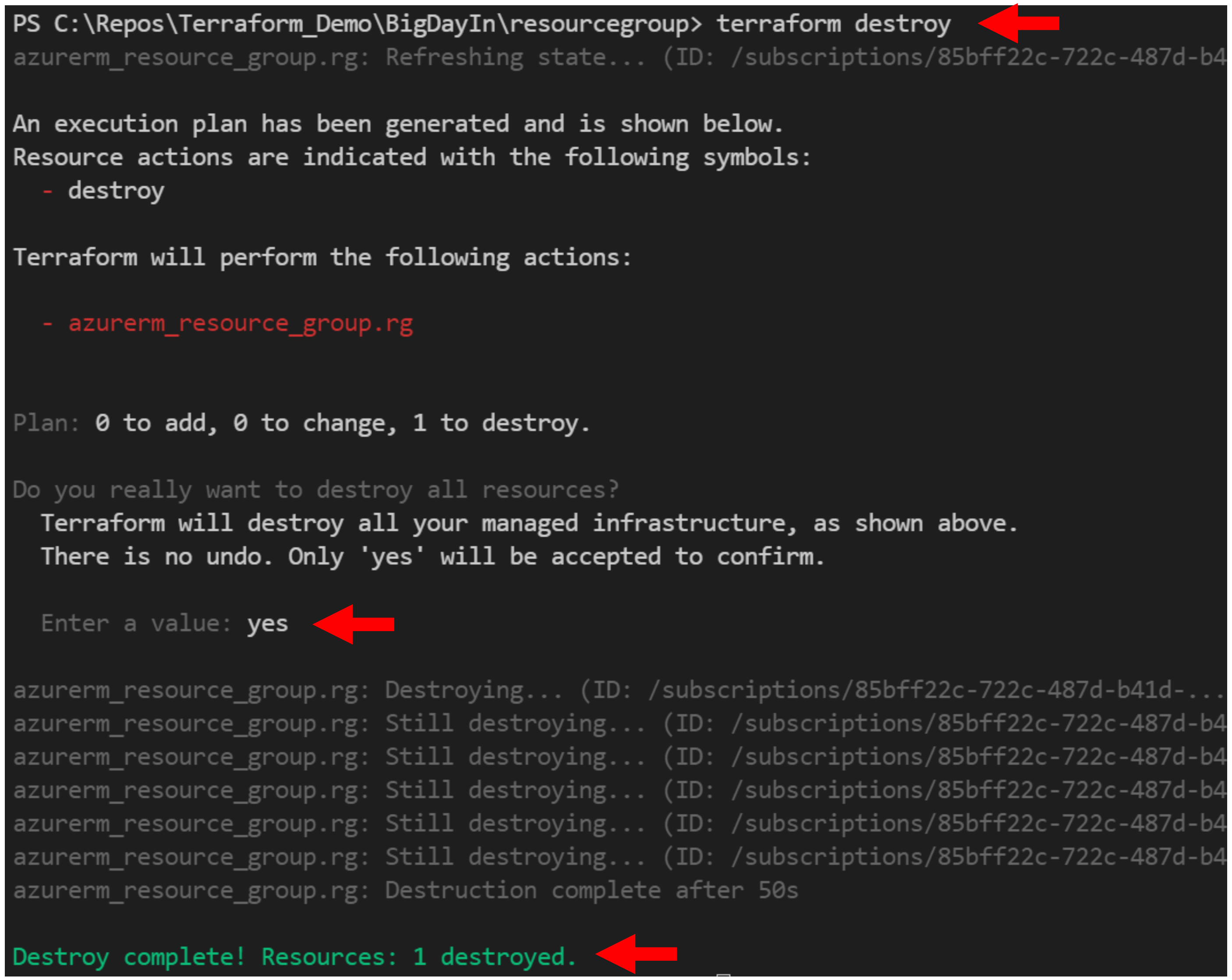
The **destroy** command will remove all resources specified in the deployment .tf files. When executed, a plan is printed on screen showing the resources that will be destroyed, and you are prompted for confirmation.

To facilitate more automated deployments the ***-auto-approve*** parameter can be specified to skip the confirmation step.

Where required, specify the ***–var*** and ***–var-file*** parameters to pass in variable values to the **destroy** command.

This command can take some time to run for large deployments. Continuous output will be provided to assist in monitoring progress.

Further Information:
[Terraform Destroy Command]



```
PS C:\Repos\Terraform_Demo\BigDayIn\resourcegroup> terraform destroy
azurerm_resource_group.rg: Refreshing state... (ID: /subscriptions/85bff22c-722c-487d-b4

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  - destroy

Terraform will perform the following actions:

  - azurerm_resource_group.rg


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

Do you really want to destroy all resources?
  Terraform will destroy all your managed infrastructure, as shown above.
  There is no undo. Only 'yes' will be accepted to confirm.

  Enter a value: yes

azurerm_resource_group.rg: Destroying... (ID: /subscriptions/85bff22c-722c-487d-b41d-...
azurerm_resource_group.rg: Still destroying... (ID: /subscriptions/85bff22c-722c-487d-b4
azurerm_resource_group.rg: Still destroying... (ID: /subscriptions/85bff22c-722c-487d-b4
azurerm_resource_group.rg: Still destroying... (ID: /subscriptions/85bff22c-722c-487d-b4
azurerm_resource_group.rg: Still destroying... (ID: /subscriptions/85bff22c-722c-487d-b4
azurerm_resource_group.rg: Still destroying... (ID: /subscriptions/85bff22c-722c-487d-b4
azurerm_resource_group.rg: Destruction complete after 50s

Destroy complete! Resources: 1 destroyed.
```

Microsoft

Terraform on Azure DevOps

Why use Terraform?

- Cleaner Code - Easier to use !!
  - Not JSON - Less nesting, less ([{brackets}])
  - No API version specified
  - Supports inline comments!

- Implicit dependencies vs arm - Dependencies must be explicitly defined using nesting, or DependsOn property

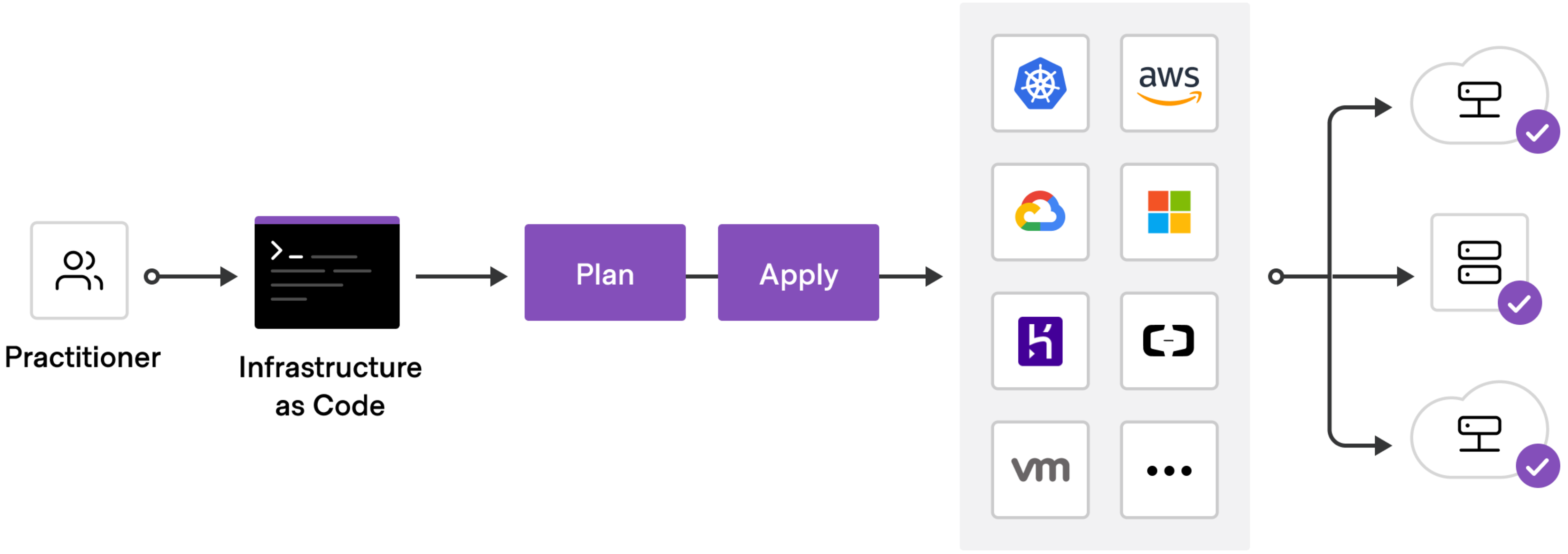
- One language and Cross Platform – for customers using a multi cloud strategy.

Caveats and Issues

- Very new, or advanced Azure functionality may not be natively available in Terraform Resource
  - Workarounds do exist !!

- OSS project - under very active development.
  - Relatively new project – started in 2014
  - If you encounter issues, submit bugs and feature requests via Github!

Demo



Practitioner
Infrastructure as Code
Plan
Apply
aws
vm
•••