Managing state Lab 1.8

Common lifecycle management areas that deal with state with refresh, ignore, move and taint.

Overview

One of Terraform's strengths is lifecycle management. Knowing how to work with the Terraform state is important.

In this lab you will

- refresh the local state file
- learn how to tolerate certain changes using lifecycle ignore
- fix Terraform identification issues with move
- taint a single resource to force a recreation

Starting point

Your files should look similar to this:

- provider.tf

```
terraform {
  required_providers {
    azurerm = {
      source  = "hashicorp/azurerm"
      version = "~>3.1"
    }
  }
}

provider "azurerm" {
  features {}

  storage_use_azuread = true
}
```

- variables.tf

```
variable "resource_group_name" {
  description = "Name for the resource group"
  type        = string
  default     = "terraform-basics"
}

```

```
variable "location" {
  description = "Azure region"
  type        = string
  default     = "West Europe"
}

variable "container_group_name" {
  description = "Name of the container group"
  type        = string
  default     = "terraform-basics"
}
```

main.tf

```
locals {
  uniq = substr(sha1(azurerm_resource_group.basics.id), 0, 8)
}

resource "azurerm_resource_group" "basics" {
  name     = var.resource_group_name
  location = var.location
}

resource "azurerm_container_group" "example" {
  name                = var.container_group_name
  location            = azurerm_resource_group.basics.location
  resource_group_name = azurerm_resource_group.basics.name
  ip_address_type     = "Public"
  dns_name_label      = "${var.container_group_name}-${local.uniq}"
  os_type             = "Linux"

  container {
    name   = "inspectorgadget"
    image  = "jelledruyts/inspectorgadget:latest"
    cpu    = "0.5"
    memory = "1.0"

    ports {
      port     = 80
      protocol = "TCP"
    }
  }
}
```

outputs.tf

```
output "ip_address" {
  value = azurerm_container_group.example.ip_address
}

output "fqdn" {
  value = "http://${azurerm_container_group.example.fqdn}"
}
```

- terraform.tfvars
- `location = "UK South"`
  You may have set a different value for *location*.

Refresh

Running `terraform plan` or `terraform apply` forces the azurerm provider to communicate with Azure to get the current state. This is stored in memory for the comparison ("diff") against the config files and determine what (if anything) needs to be done.
You can always update the local state file (terraform.tfstate) using `terraform refresh`.
Let's see it in action.

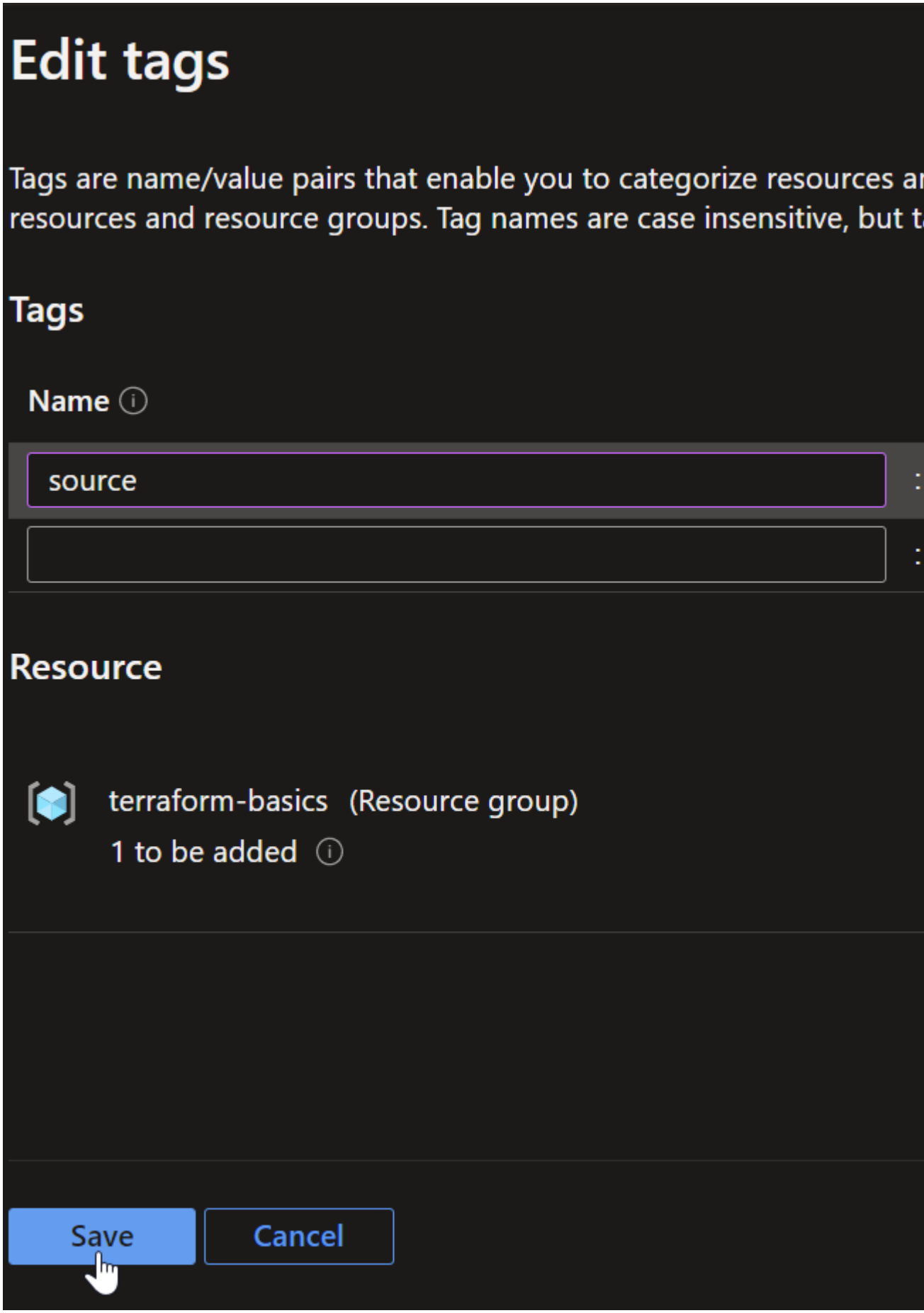
1. Check the current state for the resource group
2. `terraform state show azurerm_resource_group.basics`

   Example output:

```
# azurerm_resource_group.basics:

resource "azurerm_resource_group" "basics" {

    id       = "/subscriptions/2ca40be1-7e80-4f2b-92f7-
06b2123a68cc/resourceGroups/terraform-basics"

    location = "uksouth"

    name     = "terraform-basics"

    tags     = {}

}
```

3. Add a tag in the portal

   Open the Azure portal, find the resource group and add a tag: **source = terraform**


Edit tags
Tags are name/value pairs that enable you to categorize resources a
resources and resource groups. Tag names are case insensitive, but t
Tags
Name
source
Resource
terraform-basics (Resource group)
1 to be added
Save
Cancel

4. Redisplay the state
5. `terraform state show azurerm_resource_group.basics`

   Unsurprisingly, the output is unchanged. It is only a JSON text file.

6. Refresh the state file
7. `terraform refresh`

   Example output:

```
azurerm_resource_group.basics: Refreshing state...
[id=/subscriptions/2ca40be1-7e80-4f2b-92f7-
06b2123a68cc/resourceGroups/terraform-basics]

azurerm_container_group.example: Refreshing state...
[id=/subscriptions/2ca40be1-7e80-4f2b-92f7-
06b2123a68cc/resourceGroups/terraform-
basics/providers/Microsoft.ContainerInstance/containerGroups/terraform-
basics]



Outputs:



fqdn = "http://terraform-basics-c3818179.uksouth.azurecontainer.io"

ip_address = "20.108.130.109"
```

8. Display the updated state
9. `terraform state show azurerm_resource_group.basics`

   Example output:

```
# azurerm_resource_group.basics:

resource "azurerm_resource_group" "basics" {

    id       = "/subscriptions/2ca40be1-7e80-4f2b-92f7-
06b2123a68cc/resourceGroups/terraform-basics"

    location = "uksouth"

    name     = "terraform-basics"

    tags     = {

        "source" = "terraform"

    }

}
```

The state file is now up to date. It can be beneficial for state to be kept current, particularly if you are using read only remote states or extracting values via scripting.

You may find that running a `terraform plan` soon after a `terraform apply` shows some additional detail that is not in state such as the formation of empty arrays and lists etc. The plan will alert you to this and prompts you to run `terraform apply --refresh-only` to fully sync up.

Handling changes

Ideally, the resource groups managed by Terraform will not be subject to manual changes. However, in the real world this is a common occurrence and you may need to update the config to handle it.

Let's use a common example, to see the impact when someone adds a new tag.

1. Run a plan
2. `terraform plan`

   Example output:

```
azurerm_resource_group.basics: Refreshing state...
[id=/subscriptions/2ca40be1-7e80-4f2b-92f7-
06b2123a68cc/resourceGroups/terraform-basics]

azurerm_container_group.example: Refreshing state...
[id=/subscriptions/2ca40be1-7e80-4f2b-92f7-
06b2123a68cc/resourceGroups/terraform-
basics/providers/Microsoft.ContainerInstance/containerGroups/terraform-
basics]



Terraform used the selected providers to generate the following execution

plan. Resource actions are indicated with the following symbols:

  ~ update in-place



Terraform will perform the following actions:



  # azurerm_resource_group.basics will be updated in-place

  ~ resource "azurerm_resource_group" "basics" {
```

```
        id       = "/subscriptions/2ca40be1-7e80-4f2b-92f7-
06b2123a68cc/resourceGroups/terraform-basics"

        name     = "terraform-basics"

      ~ tags     = {

          - "source" = "terraform" -> null

        }

        # (1 unchanged attribute hidden)

    }


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

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


Note: You didn't use the -out option to save this plan, so Terraform can't

guarantee to take exactly these actions if you run "terraform apply" now.
```

If you were to run `terraform apply` now, then Terraform would remove the *source* tag and make sure the environment matches the definition in the config files.

Example updated resource block:

```
resource "azurerm_resource_group" "basics" {
  name     = var.resource_group_name
  location = var.location

  tags = {
    source = "terraform"
  }
}
```

2. Check for a clean plan

3. `terraform plan`

   Example output:

```
azurerm_resource_group.basics: Refreshing state...
[id=/subscriptions/2ca40be1-7e80-4f2b-92f7-
06b2123a68cc/resourceGroups/terraform-basics]

azurerm_container_group.example: Refreshing state...
[id=/subscriptions/2ca40be1-7e80-4f2b-92f7-
06b2123a68cc/resourceGroups/terraform-
basics/providers/Microsoft.ContainerInstance/containerGroups/terraform-
basics]



No changes. Your infrastructure matches the configuration.



Terraform has compared your real infrastructure against your configuration

and found no differences, so no changes are needed.
```

Ignore

The other approach is to force Terraform to ignore certain resource attributes using lifecycle blocks.S

1. Revert the resource group block

   Remove the tags argument in the resource group block.

   Reverted resource group block:

```
resource "azurerm_resource_group" "basics" {
  name     = var.resource_group_name
  location = var.location
}
```

   A `terraform plan` would now display a planned in-place update.

2. Ignore changes to tags

   Add in a lifecycle ignore block to the resource group.

```
resource "azurerm_resource_group" "basics" {
  name     = var.resource_group_name
  location = var.location

  lifecycle {
    ignore_changes = [
      tags,
    ]
  }
}
```

   It is very common to see ignore blocks for tags. Tags are commonly updated manually, or via Azure Policies with modify effects.

   Another example if Azure Application Gateway if being used as an Application Gateway Ingress Controller (AGIC) by AKS. In this configuration the App Gateway is reconfigured dynamically using Kubernetes annotations.

3. Confirm that no changes will be made

4. `terraform plan`

   Example output:

```
azurerm_resource_group.basics: Refreshing state...
[id=/subscriptions/2ca40be1-7e80-4f2b-92f7-
06b2123a68cc/resourceGroups/terraform-basics]

azurerm_container_group.example: Refreshing state...
[id=/subscriptions/2ca40be1-7e80-4f2b-92f7-
06b2123a68cc/resourceGroups/terraform-
basics/providers/Microsoft.ContainerInstance/containerGroups/terraform-
basics]



No changes. Your infrastructure matches the configuration.



Terraform has compared your real infrastructure against your configuration

and found no differences, so no changes are needed.
```

   The process where the plan creates an in-memory state from the provider calls and then compares against the config files is called a *diff*. The ignore statements specifies any attributes to be excluded from the diff.

Renaming

Sometimes you need to tweak the Terraform identifiers. It may be a straight rename, a shift from a single resource to using *count* or *for_each* or moving something to and from a module. This section will go through a simple example.

1. Check for a clean plan
2. `terraform plan`

   Example output:

```
azurerm_resource_group.basics: Refreshing state...
[id=/subscriptions/2ca40be1-7e80-4f2b-92f7-
06b2123a68cc/resourceGroups/terraform-basics]

azurerm_container_group.example: Refreshing state...
[id=/subscriptions/2ca40be1-7e80-4f2b-92f7-
06b2123a68cc/resourceGroups/terraform-
basics/providers/Microsoft.ContainerInstance/containerGroups/terraform-
basics]



No changes. Your infrastructure matches the configuration.



Terraform has compared your real infrastructure against your configuration

and found no differences, so no changes are needed.
```

3. List out the identifiers in state
4. `terraform state list`

   Example output:

```
azurerm_container_group.example

azurerm_resource_group.basics
```

   We will change
   the *azurerm_container_group.example* to *azurerm_container_group.basics*.

5. Update main.tf

   Change the label for the azurerm_container_group identifier
   from **“example”** to **“basics”**.

6. Rerun plan
7. `terraform plan`

   You should see validation errors. Refactor the two outputs.

8. Rerun plan

```
9. terraform plan
```

   You should see the container group will be deleted and recreated.

If so, then use `terraform taint` to force the resource to be recreated.

1. Check for a clean plan
2. `terraform plan`

   Example output:

```
azurerm_resource_group.basics: Refreshing state...
[id=/subscriptions/2ca40be1-7e80-4f2b-92f7-
06b2123a68cc/resourceGroups/terraform-basics]

azurerm_container_group.basics: Refreshing state...
[id=/subscriptions/2ca40be1-7e80-4f2b-92f7-
06b2123a68cc/resourceGroups/terraform-
basics/providers/Microsoft.ContainerInstance/containerGroups/terraform-
basics]



No changes. Your infrastructure matches the configuration.



Terraform has compared your real infrastructure against your configuration

and found no differences, so no changes are needed.
```

3. List out the identifiers
4. `terraform state list`

   Example output:

```
azurerm_container_group.basics

azurerm_resource_group.basics
```

5. Taint the container group

   Force the container group to be recreated as an example.

```
terraform taint azurerm_container_group.basics
Resource instance azurerm_container_group.basics has been marked as
tainted.
```

6. Plan
7. `terraform plan`

   Example output:

```
azurerm_resource_group.basics: Refreshing state...
[id=/subscriptions/2ca40be1-7e80-4f2b-92f7-
06b2123a68cc/resourceGroups/terraform-basics]
```

```
azurerm_container_group.basics: Refreshing state...
[id=/subscriptions/2ca40be1-7e80-4f2b-92f7-
06b2123a68cc/resourceGroups/terraform-
basics/providers/Microsoft.ContainerInstance/containerGroups/terraform-
basics]



Terraform used the selected providers to generate the following execution

plan. Resource actions are indicated with the following symbols:

-/+ destroy and then create replacement



Terraform will perform the following actions:



  # azurerm_container_group.basics is tainted, so must be replaced

-/+ resource "azurerm_container_group" "basics" {

      ~ exposed_port        = [

          - {

              - port     = 80

              - protocol = "TCP"

            },

        ] -> (known after apply)

      ~ fqdn                = "terraform-basics-
c3818179.uksouth.azurecontainer.io" -> (known after apply)

      ~ id                  = "/subscriptions/2ca40be1-7e80-4f2b-92f7-
06b2123a68cc/resourceGroups/terraform-
basics/providers/Microsoft.ContainerInstance/containerGroups/terraform-
basics" -> (known after apply)

      ~ ip_address          = "20.108.130.109" -> (known after apply)

      ~ ip_address_type     = "Public" -> "public"

        name                = "terraform-basics"

      - tags                = {} -> null

        # (5 unchanged attributes hidden)



      ~ container {
```

```
          ~ commands                       = [] -> (known after apply)
          - environment_variables          = {} -> null
            name                           = "inspectorgadget"
          - secure_environment_variables = (sensitive value)
            # (3 unchanged attributes hidden)


            # (1 unchanged block hidden)
        }
    }


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


Changes to Outputs:
  ~ fqdn       = "http://terraform-basics-
c3818179.uksouth.azurecontainer.io" -> (known after apply)
  ~ ip_address = "20.108.130.109" -> (known after apply)

──────────────────────────────────────────────────────────────────────────────
──


Note: You didn't use the -out option to save this plan, so Terraform can't
guarantee to take exactly these actions if you run "terraform apply" now.
```

8. Apply
9. `terraform apply --auto-approve`

   The Azure Container Instance will be recreated.

Summary

Terraform can make life simpler in terms of lifecycle management and seeing the planned impact of configuration changes, but it is useful to know how to use the tools to manage these scenarios.

In the next lab we will handle the import of a resource that has been created outside of Terraform and bring it into state safely.
