Lab: Terraform Resource Lifecycles

Terraform has the ability to support the parallel management of resources because of it's resource graph allowing it to optimize the speed of deployments. The resource graph dictates the order in which Terraform creates and destroys resources, and this order is typically appropriate. There are however situations where the we wish to change the default lifecycle behavior that Terraform uses.

To provide you with control over dependency errors, Terraform has a `lifecycle` block. This lab demonstrates how to use lifecycle directives to control the order in which Terraform creates and destroys resources.

- Task 1: Use `create_before_destroy` with a simple AWS security group and instance
- Task 2: Use `prevent_destroy` with an instance

Task 1: Use `create_before_destroy` with a simple AWS security group and instance

Terraform's default behavior when marking a resource to be replaced is to first destroy the resource and then create it. If the destruction succeeds cleanly, then and only then are replacement resources created. To alter this order of operation we can utilize the lifecycle directive `create_before_destroy` which does what it says on the tin. Instead of destroying an instance and then provisioning a new one with the specified attributes, it will provision first. So two instances will exist simultaneously, then the other will be destroyed.

Let's create a simple AWS configuration with a security group and an associated EC2 instance. Provision them with `terraform`, then make a change to the security group. Observe that `apply` fails because the security group can not be destroyed and recreated while the instance lives.

You'll solve this situation by using `create_before_destroy` to create the new security group before destroying the original one.

1.1: Add a new security group to the security_groups list of our server module

Add a new security group to the `security_groups` list of our server module by including `aws_security_group.main.id`

```
module "server_subnet_1" {
  source          = "./modules/web_server"
  ami             = data.aws_ami.ubuntu.id
  key_name        = aws_key_pair.generated.key_name
```

```
  user            = "ubuntu"
  private_key     = tls_private_key.generated.private_key_pem
  subnet_id       = aws_subnet.public_subnets["public_subnet_1"].id
  security_groups = [aws_security_group.vpc-ping.id, aws_security_group.
     ingress-ssh.id, aws_security_group.vpc-web.id, aws_security_group.
     main.id]
}
```

Initialize and apply the change to add the security group to our server module's `security_groups` list.

```
terraform init
terraform apply
```

1.2: Change the name of the security group

In order to see how some resources cannot be recreated under the default `lifecyle` settings, let's attempt to change the name of the security group from `core-sg` to something like `core-sg-global`.

```
resource "aws_security_group" "main" {
  name = "core-sg-global"

  vpc_id = aws_vpc.vpc.id

  dynamic "ingress" {
    for_each = var.web_ingress
    content {
      description = ingress.value.description
      from_port   = ingress.value.port
      to_port     = ingress.value.port
      protocol    = ingress.value.protocol
      cidr_blocks = ingress.value.cidr_blocks
    }
  }
}
```

Apply this change.

```
terraform apply
```

```
Terraform used the selected providers to generate the following execution
    plan. Resource actions are indicated with the following symbols:
  ~ update in-place
-/+ destroy and then create replacement
```

```
Terraform will perform the following actions:

  # aws_security_group.main must be replaced
-/+ resource "aws_security_group" "main" {
      ~ arn                    = "arn:aws:ec2:us-east-1:508140242758:
         security-group/sg-00157499a6de61832" -> (known after apply)
      ~ egress                 = [] -> (known after apply)
      ~ id                     = "sg-00157499a6de61832" -> (known after
         apply)
      ~ name                   = "core-sg" -> "core-sg-global" # forces
         replacement
      + name_prefix            = (known after apply)
      ~ owner_id               = "508140242758" -> (known after apply)
      - tags                   = {} -> null
      ~ tags_all               = {} -> (known after apply)
        # (4 unchanged attributes hidden)
    }

  # module.server_subnet_1.aws_instance.web will be updated in-place
  ~ resource "aws_instance" "web" {
        id                                   = "i-0fbb3100e8671a855"
        tags                                 = {
            "Environment" = "Training"
            "Name"        = "Web Server from Module"
        }
      ~ vpc_security_group_ids               = [
          - "sg-00157499a6de61832",
          - "sg-00dc379cbd0ad7332",
          - "sg-01fb306fc93cb941c",
          - "sg-0e0544dac3596af26",
        ] -> (known after apply)
        # (28 unchanged attributes hidden)





        # (5 unchanged blocks hidden)
    }

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

Notice that the default Terraform behavior is to destroy then create this resource is shown by the `-/+ destroy and then create replacement` statement.

Proceed with the apply.

```
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
```

NOTE: This action takes many minutes and eventually shows an error. You may choose to terminate the `apply` action with `^C` before the 15 minutes elapses. You may have to terminate twice to exit.

```
aws_security_group.main: Destroying... [id=sg-00157499a6de61832]
aws_security_group.main: Still destroying... [id=sg-00157499a6de61832, 10s
     elapsed]
aws_security_group.main: Still destroying... [id=sg-00157499a6de61832, 20s
     elapsed]
aws_security_group.main: Still destroying... [id=sg-00157499a6de61832, 30s
     elapsed]

....

aws_security_group.main: Still destroying... [id=sg-00157499a6de61832, 14
    m40s elapsed]
aws_security_group.main: Still destroying... [id=sg-00157499a6de61832, 14
    m50s elapsed]
aws_security_group.main: Still destroying... [id=sg-00157499a6de61832, 15
    m0s elapsed]

| Error: Error deleting security group: DependencyViolation: resource sg
    -00157499a6de61832 has a dependent object
|       status code: 400, request id: 80dc904d-9439-41eb-8574-4b173685e72f
|
|
```

This is occuring because we have other resources that are dependent on this security group, and therefore the default Terraform behavior of destroying and then recreating the new security group is causing a dependency violation. We can solve this by using the `create_before_destroy` lifecycle directive to tell Terraform to first create the new security group before destroying the original.

1.3: Use `create_before_destroy`

Add a `lifecycle` configuration block to the `aws_security_group` resource. Specify that this resource should be created before the existing security group is destroyed.

```
resource "aws_security_group" "main" {
  name = "core-sg-global"

  vpc_id = aws_vpc.vpc.id
  dynamic "ingress" {
    for_each = var.web_ingress
    content {
      description = ingress.value.description
      from_port   = ingress.value.port
      to_port     = ingress.value.port
      protocol    = ingress.value.protocol
      cidr_blocks = ingress.value.cidr_blocks
    }
  }

  lifecycle {
    create_before_destroy = true
  }

}
```

Now provision the new resources with the improved `lifecycle` configuration.

```
terraform apply
```

```
Terraform used the selected providers to generate the following execution
    plan. Resource actions are
indicated with the following symbols:
  ~ update in-place
+/- create replacement and then destroy

Terraform will perform the following actions:

  # aws_security_group.main must be replaced
+/- resource "aws_security_group" "main" {
      ~ arn                    = "arn:aws:ec2:us-east-1:508140242758:
         security-group/sg-00157499a6de61832" -> (known after apply)
      ~ egress                 = [] -> (known after apply)
      ~ id                     = "sg-00157499a6de61832" -> (known after
         apply)
      ~ name                   = "core-sg" -> "core-sg-global" # forces
         replacement
      + name_prefix            = (known after apply)
      ~ owner_id               = "508140242758" -> (known after apply)
      - tags                   = {} -> null
      ~ tags_all               = {} -> (known after apply)
        # (4 unchanged attributes hidden)
    }

  # module.server_subnet_1.aws_instance.web will be updated in-place
  ~ resource "aws_instance" "web" {
```

```
        id                                     = "i-0fbb3100e8671a855"
        tags                                   = {
            "Environment" = "Training"
            "Name"        = "Web Server from Module"
        }
      ~ vpc_security_group_ids                 = [
          - "sg-00157499a6de61832",
          - "sg-00dc379cbd0ad7332",
          - "sg-01fb306fc93cb941c",
          - "sg-0e0544dac3596af26",
        ] -> (known after apply)
        # (28 unchanged attributes hidden)





        # (5 unchanged blocks hidden)
    }

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

Notice now that the Terraform behavior is to create then destroy this resource is shown by the +/- `create replacement and then destroy` statement.

Proceed with the apply.

```
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
```

It should now succeed within a short amount of time as the new security group is created frist, applied to our server and then the old security group is destroyed. Using the lifecyle block we controlled the order in which Terraform creates and destroys resources, removing the dependency violation of renaming a security group.

```
aws_security_group.main: Creating...
aws_security_group.main: Creation complete after 4s [id=sg-015
    f1eaa2c3f29f4c]
module.server_subnet_1.aws_instance.web: Modifying... [id=i-0
    fbb3100e8671a855]
module.server_subnet_1.aws_instance.web: Modifications complete after 5s [
    id=i-0fbb3100e8671a855]
aws_security_group.main (deposed object 5c96d2e2): Destroying... [id=sg
    -00157499a6de61832]
```

```
aws_security_group.main: Destruction complete after 1s

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

Task 2: Use `prevent_destroy` with an instance

Another lifecycle directive that we may wish to include in our configuraiton is `prevent_destroy`. This warns if any change would result in destroying a resource. All resources that this resource depends on must also be set to `prevent_destroy`. We'll demonstrate how `prevent_destroy` can be used to guard an instance from being destroyed.

2.1: Use `prevent_destroy`

Add `prevent_destroy = true` to the same `lifecycle` stanza where you added `create_before_destroy`
.

```
resource "aws_security_group" "main" {
  name = "core-sg-global"

  # ...

  lifecycle {
    create_before_destroy = true
    prevent_destroy       = true
  }
}
```

Attempt to destroy the existing infrastructure. You should see the error that follows.

```
terraform destroy -auto-approve
```

```

Error: Instance cannot be destroyed

  on main.tf line 378:
  378: resource "aws_security_group" "main" {

 Resource aws_security_group.main has lifecycle.prevent_destroy set, but
     the plan calls for this resource to be destroyed. To avoid this error
     and continue with the plan, either disable lifecycle.prevent_destroy
     or reduce the scope of the plan using the -target flag.
```

2.2: Destroy cleanly

Now that you have finished the steps in this lab, destroy the infrastructure you have created.

Remove the `prevent_destroy` attribute.

```
resource "aws_security_group" "main" {
  name = "core-sg-global"

  # ...

  lifecycle {
    create_before_destroy = true
    # Comment out or delete this line
    # prevent_destroy = true
  }
}
```

Finally, run `destroy`.

```
terraform destroy -auto-approve
```

The command should succeed and you should see a message confirming `Destroy complete`!

The `prevent_destroy` lifecycle directive can be used on resources that are stateful in nature (s3 buckets, RDS instances, long lived VMs, etc.) as a mechanism to help prevent accidently destroying items that have long lived data within them.