Terraform Operations

Raúl Estrada

Octubre 2020

Operations

We shall examine five main TF operations:

- Validating a template
- Testing (dry-run)
- Initial deployment
- Updating a deployment
- Removal of a deployment

Validation

Before going any further, a basic syntax check should be done with the `terraform validate` command. After renaming one of the variables in `resources.tf`, validate returns an `unknown variable` error:

Copy

```
$ terraform validate
Error validating: 1 error(s) occurred:
* provider config 'aws': unknown variable referenced: 'aws-region-1'. define it with 'variable' blocks
```

Once the variable name has been corrected, re-running `validate` returns no output, meaning validation has passed.

Dry-run

The next step is to perform a test/dry-run execution with `terraform plan`, which displays what would happen during an actual deployment. The command returns a color-coded list of resources and their properties or more precisely, as follows:

Copy

```
$ terraform plan
Resources are shown in alphabetical order for quick scanning. Green resources will be created (or destroyed
```

To literally get the picture of what the to-be-deployed infrastructure looks like, you could use `terraform graph` :

Copy

```
$ terraform graph > my_graph.dot
```

DOT files can be manipulated with the **Graphviz** open source software (Please see http://www.graphviz.org) or many online readers/converters. The following diagram is a portion of a larger graph representing the template we designed earlier:

Terraform graph

Deployment

If you are happy with the plan and graph, the template can now be deployed using `terraform apply` :

Copy

```
$ terraform apply
aws_eip.nat-eip: Creating...
allocation_id: "" => "<computed>"
association_id: "" => "<computed>"
domain: "" => "<computed>"
instance: "" => "<computed>"
network_interface: "" => "<computed>"
private_ip: "" => "<computed>"
public_ip: "" => "<computed>"
vpc: "" => "1"
aws_vpc.terraform-vpc: Creating...
cidr_block: "" => "10.0.0.0/16"
default_network_acl_id: "" => "<computed>"
default_security_group_id: "" => "<computed>"
dhcp_options_id: "" => "<computed>"
enable_classiclink: "" => "<computed>"
enable_dns_hostnames: "" => "<computed>"
Apply complete! Resources: 22 added, 0 changed, 0 destroyed.
```

The state of your infrastructure has been saved to the following path. This state is required to modify and destroy your infrastructure, so keep it safe. To inspect the complete state, use the `terraform show` command.

Copy

```
State path: terraform.tfstate
Outputs:
ELB URI = terraform-elb-xxxxxx.us-east-1.elb.amazonaws.com
NAT EIP = x.x.x.x
RDS Endpoint = terraform-rds.xxxxxx.us-east-1.rds.amazonaws.com:5432
VPC ID = vpc-xxxxxx
```

At the end of a successful deployment, you will notice the `Outputs` we configured earlier and a message about another important part of **TF - the state file** (please refer to https://www.terraform.io/docs/state/):

TF stores the state of your managed infrastructure from the last time TF was run. By default, this state is stored in a local file named `terraform.tfstate`, but it can also be stored remotely, which works better in a team environment.

TF uses this local state to create plans and make changes to your infrastructure. Prior to any operation, TF does a refresh to update the state with the real infrastructure.

In a sense, the `state` file contains a snapshot of your infrastructure and is used to calculate any changes when a template has been modified. Normally, you would keep the `terraform.tfstate` file under version control alongside your templates. In a team environment however, if you encounter too many merge conflicts you can switch to storing the `state` file(s) in an alternative location such as S3 (please see: https://www.terraform.io/docs/state/remote/index.html).

Allow a few minutes for the EC2 node to fully initialize, then try loading the ELB URI from the preceding `Outputs` in your browser. You should be greeted by **nginx**, as shown in the following screenshot:

Updates

As per Murphy 's Law, as soon as we deploy a template, a change to it will become necessary. Fortunately, all that is needed for this is to update and re-deploy the given template.

Let's say we need to add a new rule to the ELB security group (shown in bold):

1. Update the `"aws_security_group" "terraform-elb"` resource block in `resources.tf` :

Copy

```
resource "aws_security_group" "terraform-elb" {
name = "terraform-elb"
description = "ELB security group"
vpc_id = "${aws_vpc.terraform-vpc.id}"

ingress {
from_port = "80"
to_port = "80"
protocol = "tcp"
cidr_blocks = ["0.0.0.0/0"]
}

ingress {
from_port = "443"
to_port = "443"
protocol = "tcp"
cidr_blocks = ["0.0.0.0/0"]
}

egress {
```

2 Verify what is about to change:

Copy

```
$ terraform plan
...
~ aws_security_group.terraform-elb
ingress.#: "1" => "2"
ingress.2214680975.cidr_blocks.#: "1" => "1"
ingress.2214680975.cidr_blocks.0: "0.0.0.0/0" => "0.0.0.0/0"
ingress.2214680975.from_port: "80" => "80"
ingress.2214680975.protocol: "tcp" => "tcp"
ingress.2214680975.security_groups.#: "0" => "0"
ingress.2214680975.self: "0" => "0"
ingress.2214680975.to_port: "80" => "80"
ingress.2617001939.cidr_blocks.#: "0" => "1"
ingress.2617001939.cidr_blocks.0: "" => "0.0.0.0/0"
ingress.2617001939.from_port: "" => "443"
ingress.2617001939.protocol: "" => "tcp"
ingress.2617001939.security_groups.#: "0" => "0"
ingress.2617001939.self: "" => "0"
ingress.2617001939.to_port: "" => "443"
Plan: 0 to add, 1 to change, 0 to destroy.
```

3 Deploy the change:

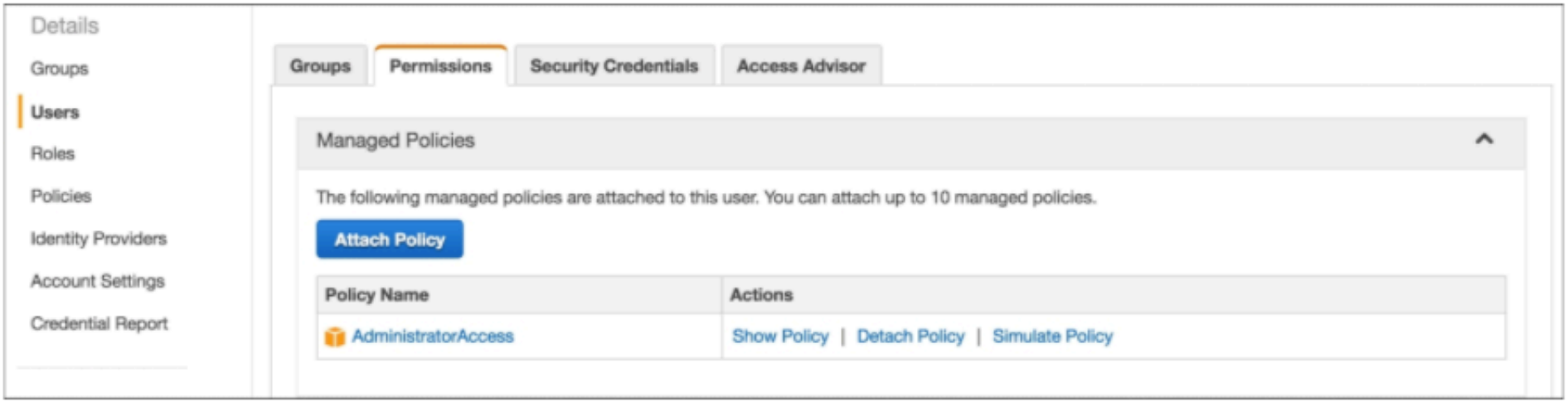
Copy

```
$ terraform apply
...
aws_security_group.terraform-elb: Modifying...
ingress.#: "1" => "2"
ingress.2214680975.cidr_blocks.#: "1" => "1"
ingress.2214680975.cidr_blocks.0: "0.0.0.0/0" => "0.0.0.0/0"
ingress.2214680975.from_port: "80" => "80"
ingress.2214680975.protocol: "tcp" => "tcp"
ingress.2214680975.security_groups.#: "0" => "0"
ingress.2214680975.self: "0" => "0"
ingress.2214680975.to_port: "80" => "80"
ingress.2617001939.cidr_blocks.#: "0" => "1"
ingress.2617001939.cidr_blocks.0: "" => "0.0.0.0/0"
ingress.2617001939.from_port: "" => "443"
ingress.2617001939.protocol: "" => "tcp"
ingress.2617001939.security_groups.#: "0" => "0"
ingress.2617001939.self: "" => "0"
ingress.2617001939.to_port: "" => "443"
aws_security_group.terraform-elb: Modifications complete
...
```

Removal

This is a friendly reminder to always remove AWS resources after you are done experimenting with them to avoid any unexpected charges.

Before performing any `delete` operations, we will need to grant such privileges to the (`terraform`) IAM user we created in the beginning of this chapter. As a shortcut, you could temporarily attach the **AdministratorAccess** managed policy to the user via the AWS Console, as shown in the following figure:



To remove the VPC and all associated resources that we created as part of this example, we will use `terraform destroy` :

Copy

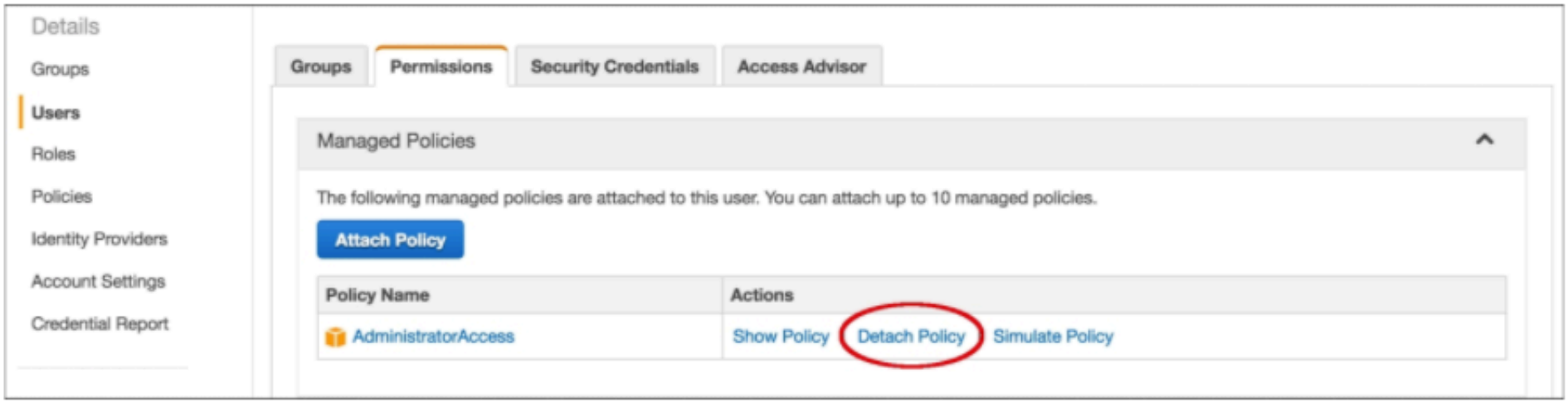
```
$ terraform destroy
Do you really want to destroy?
Terraform will delete all your managed infrastructure.
There is no undo. Only 'yes' will be accepted to confirm.
Enter a value: yes
```

`Terraform` asks for a confirmation then proceeds to destroy resources, ending with the following:

Copy

```
Apply complete! Resources: 0 added, 0 changed, 22 destroyed.
```

Next, we remove the temporary admin access we granted to the IAM user by detaching the **AdministratorAccess** managed policy, as shown in the following screenshot:



Then, verify that the VPC is no longer visible in the AWS Console.