Terraform
Template design

Raúl Estrada

Octubre 2020

Template design

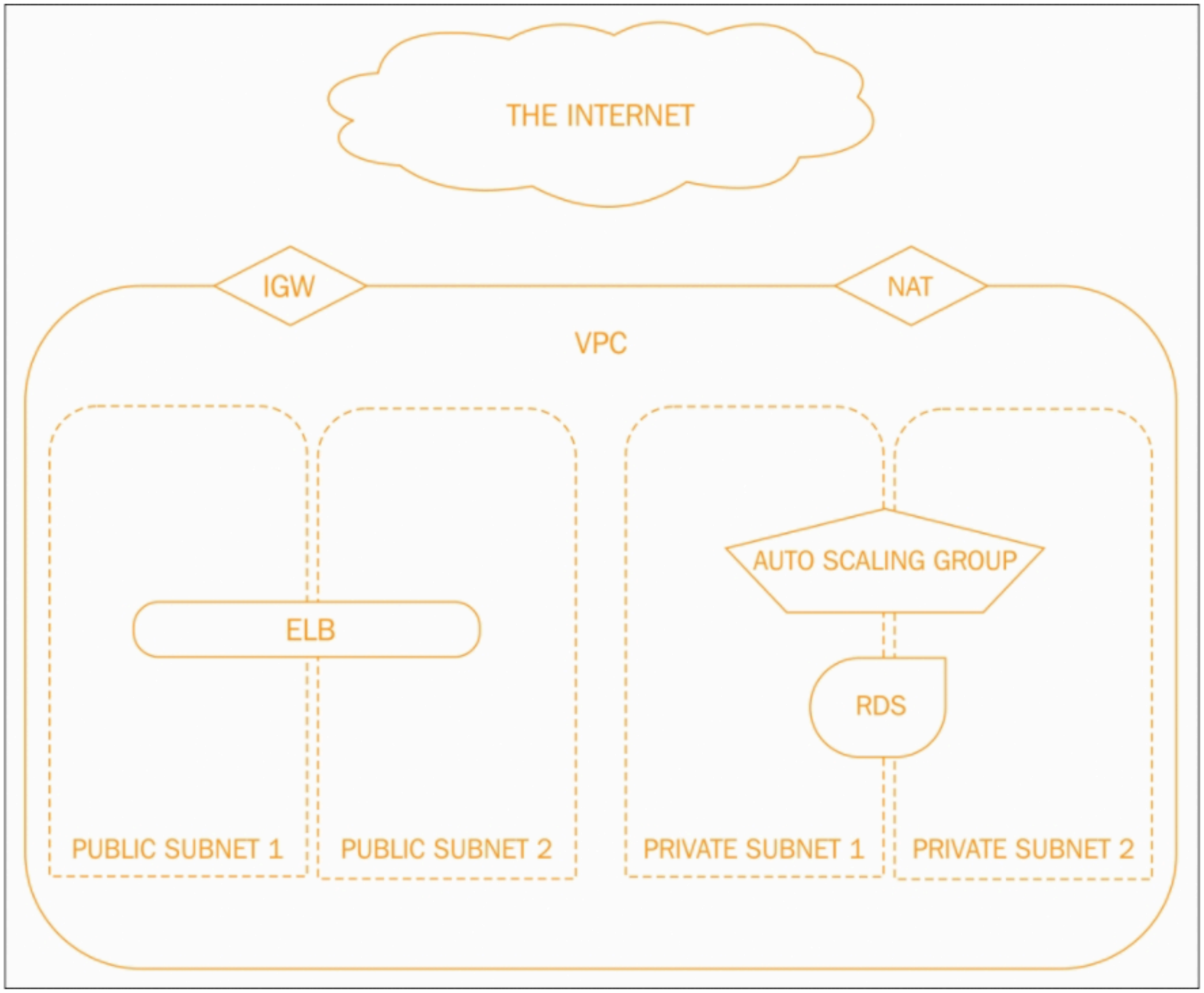
Before we get to coding, here are some of the rules:

- You could choose to write a TF template as a single large file or a combination of smaller ones
- Templates can be written in pure JSON or TF's own format
- TF will look for files with `.tf` or `.tf.json` extensions in a given folder and load them in alphabetical order
- TF templates are declarative, hence the order in which resources appear in them does not affect the flow of execution

A TF template generally consists of three sections: **resources**, **variables**, and **outputs**. As mentioned in the preceding section, it is a matter of personal preference how you arrange these; however, for better readability I suggest we make use of the TF format and write each section to a separate file. Also, while the file extensions are of importance, the filenames are up to you.

Resources

In a way, this file holds the main part of a template, as the resources represent the actual components that end up being provisioned. For example, we will be using the VPC Terraform resource, RDS, ELB and a few others to provision what roughly looks like this:


THE INTERNET
IGW
NAT
VPC
AUTO SCALING GROUP
ELB
RDS
PUBLIC SUBNET 1
PUBLIC SUBNET 2
PRIVATE SUBNET 1
PRIVATE SUBNET 2

Since template elements can be written in any order, TF determines the flow of execution by examining any references that it finds (for example, a VPC should exist before an ELB that is said to belong to it is created). Alternatively, explicit flow control attributes such as `depends_on` are used, as we will observe shortly.

To find out more, let us go through the contents of the `resources.tf` file.

First, we tell Terraform what provider to use for our infrastructure:

Copy

```
# Set a Provider
  provider "aws"
{
  region = "${var.aws-region}"
}
```

You will notice that no credentials are specified, since we set them as environment variables earlier.

Now we can add the VPC and its networking components:

Copy

```
{
  cidr_block = "${var.vpc-cidr}"

  tags
  {
    Name = "${var.vpc-name}"
  }
}

# Create an Internet Gateway
  resource "aws_internet_gateway" "terraform-igw"
{
  vpc_id = "${aws_vpc.terraform-vpc.id}"
}

# Create NAT
  resource "aws_eip" "nat-eip"
{
  vpc = true
}
```

So far, we have declared the VPC, its Internet and NAT gateways, plus a set of public and private subnets with matching routing tables.

It will help clarify the syntax if we examined some of those resource blocks, line by line:

Copy

```
resource "aws_subnet" "public-1" {
```

The first argument is the type of the resource followed by an arbitrary name:

Copy

```
vpc_id = "${aws_vpc.terraform-vpc.id}"
```

The `aws_subnet` resource named `public-1` has a `vpc_id` property, which refers to the `id` attribute of a different `aws_vpc` resource named `terraform-vpc`. Such references to other resources implicitly define the execution flow, that is to say, the VPC needs to exist before the subnet can be created:

Copy

```
cidr_block = "${cidrsubnet(var.vpc-cidr, 8, 1)}"
```

We will talk more about variables in a moment, but the format is `var.var_name as shown here` .

Here, we use the `cidrsubnet` function with the `vpc-cidr` variable, which returns a `cidr_block` to be assigned to the `public-1` subnet. Please refer to the TF documentation for this and other useful functions.

Next, we add a RDS to the VPC:

Copy

```
resource "aws_db_instance" "terraform" {
identifier = "${var.rds-identifier}"
allocated_storage = "${var.rds-storage-size}"
storage_type= "${var.rds-storage-type}"
engine = "${var.rds-engine}"
engine_version = "${var.rds-engine-version}"
instance_class = "${var.rds-instance-class}"
username = "${var.rds-username}"
password = "${var.rds-password}"
port = "${var.rds-port}"
vpc_security_group_ids = ["${aws_security_group.terraform-rds.id}"]
db_subnet_group_name = "${aws_db_subnet_group.rds.id}"
}
```

Here, we mostly see references to variables with a few calls to other resources.

Following the RDS is an ELB:

Copy

```
resource "aws_elb" "terraform-elb"
{
  name = "terraform-elb"
  security_groups = ["${aws_security_group.terraform-elb.id}"]
  subnets = ["${aws_subnet.public-1.id}",
"${aws_subnet.public-2.id}"]

listener
  {
   instance_port = 80
   instance_protocol = "http"
   lb_port = 80
   lb_protocol = "http"
  }

tags
  {
   Name = "terraform-elb"
  }
    }
```

Lastly, we define the EC2 Auto Scaling Group and related resources such as the Launch Configuration.

For the Launch Configuration we define the AMI and type of instance to be used, the name of the SSH keypair, EC2 security group(s) and the UserData to be used to bootstrap the instances:

Copy

```
resource "aws_launch_configuration" "terraform-lcfg" {
image_id = "${var.autoscaling-group-image-id}"
instance_type = "${var.autoscaling-group-instance-type}"
key_name = "${var.autoscaling-group-key-name}"
security_groups = ["${aws_security_group.terraform-ec2.id}"]
user_data = "#!/bin/bash \n set -euf -o pipefail \n exec 1> >(logger -s -t $(basename $0)) 2>&1 \n yum -y i

lifecycle {
create_before_destroy = true
}
```

The Auto Scaling Group takes the ID of the Launch Configuration, a list of VPC subnets, the min/max number of instances and the name of the ELB to attach provisioned instances to:

Copy

```
}
resource "aws_autoscaling_group" "terraform-asg" {
name = "terraform"
launch_configuration = "${aws_launch_configuration.terraform-lcfg.id}"
vpc_zone_identifier = ["${aws_subnet.private-1.id}", "${aws_subnet.private-2.id}"]
min_size = "${var.autoscaling-group-minsize}"
max_size = "${var.autoscaling-group-maxsize}"
load_balancers = ["${aws_elb.terraform-elb.name}"]
depends_on = ["aws_db_instance.terraform"]
tag {
key = "Name"
value = "terraform"
propagate_at_launch = true
}
}
```

The preceding `user_data` shell script will install and start NGINX onto the EC2 node(s).

Variables

We have made great use of variables to define our resources, making the template as re-usable as possible. Let us now look inside `variables.tf` to study these further.

Similarly to the resources list, we start with the VPC:

Copy

```
variable "aws-region" {
type = "string"
description = "AWS region"
}
variable "aws-availability-zones" {
type = "string"
description = "AWS zones"
}
variable "vpc-cidr" {
type = "string"
description = "VPC CIDR"
}
variable "vpc-name" {
type = "string"
description = "VPC name"
}
```

The syntax is as follows:

Copy

```
variable "variable_name" {
variable properties
}
```

`variable_name` is arbitrary, but needs to match relevant `var.var_name` references made in other parts of the template. For example, the `aws-region` variable will satisfy the `${var.aws-region}` reference we made earlier when describing the region of the `provider aws resource` .

We will mostly use `string` variables, but there is another useful type called **map** that can hold lookup tables. Maps are queried in a similar way to looking up values in a hash/dict (Please see: https://www.terraform.io/docs/configuration/variables.html).

Next comes RDS:

Copy

```
variable "rds-engine-version" {
type = "string"
description = "RDS version"
}
variable "rds-instance-class" {
type = "string"
description = "RDS instance class"
}
variable "rds-username" {
type = "string"
description = "RDS username"
}
variable "rds-password" {
type = "string"
description = "RDS password"
}
variable "rds-port" {
type = "string"
description = "RDS port number"
}
```

Lastly, we add our EC2 related variables:

Copy

```
variable "autoscaling-group-minsize" {
type = "string"
description = "Min size of the ASG"
}
variable "autoscaling-group-maxsize" {
type = "string"
description = "Max size of the ASG"
}
variable "autoscaling-group-image-id" {
type="string"
description = "EC2 AMI identifier"
}
variable "autoscaling-group-instance-type" {
type = "string"
description = "EC2 instance type"
}
variable "autoscaling-group-key-name" {
type = "string"
description = "EC2 ssh key name"
}
```

We now have the type and description of all our variables defined in `variables.tf` , but no values have been assigned to them yet.

TF is quite flexible with how this can be done. We could do it any of the following ways:

- Assign (default) values directly in `variables.tf:`
- variable " `aws-region` "{ `type = "string"` `description = "AWS region"` `default = 'us-east-1'` }
- Not assign a value to a variable, in which case TF will prompt for it at run time
- `*` Pass a `-var 'key=value'` argument directly to the TF command, like so:

Copy

```
-var 'aws-region=us-east-1'
```

- Store `key=value` pairs in a file
- Use environment variables prefixed with `TF_VAR` , as in `TF_VAR_ aws-region`

Using a `key=value` pairs file proves to be quite convenient within teams, as each engineer can have a private copy (excluded from revision control). If the file is named `terraform.tfvars` it will be read automatically by TF; alternatively, `-var-file` can be used on the command line to specify a different source.

Here is the content of our sample `terraform.tfvars` file:

Copy

```
autoscaling-group-image-id = "ami-08111162"
autoscaling-group-instance-type = "t2.nano"
autoscaling-group-key-name = "terraform"
autoscaling-group-maxsize = "1"
autoscaling-group-minsize = "1"
aws-availability-zones = "us-east-1b,us-east-1c"
aws-region = "us-east-1"
rds-engine = "postgres"
rds-engine-version = "9.5.2"
rds-identifier = "terraform-rds"
rds-instance-class = "db.t2.micro"
rds-port = "5432"
rds-storage-size = "5"
rds-storage-type = "gp2"
rds-username = "dbroot"
rds-password = "donotusethispassword"
vpc-cidr = "10.0.0.0/16"
vpc-name = "Terraform"
```

A point of interest is `aws-availability-zones` , as it holds multiple values that we interact with using the element and split functions, as seen in `resources.tf` .

Outputs

The third, mostly informational part of our template contains the TF Outputs. These allow selected values to be returned to the user when testing, deploying or after a template has been deployed. The concept is similar to how echo statements are commonly used in shell scripts to display useful information during execution.

Let us add outputs to our template by creating an `outputs.tf` file:

Copy

```
output "VPC ID" {
value = "${aws_vpc.terraform-vpc.id}"
}

output "NAT EIP" {
value = "${aws_nat_gateway.terraform-nat.public_ip}"
}

output "ELB URI" {
value = "${aws_elb.terraform-elb.dns_name}"
}
output "RDS Endpoint" {
value = "${aws_db_instance.terraform.endpoint}"
}
```

To configure an output, you simply reference a given resource and its attribute. As shown in preceding code, we have chosen the ID of the VPC, the Elastic IP address of the NAT gateway, the DNS name of the ELB and the endpoint address of the RDS instance.

This section completes the template in this example. You should now have four files in your template folder: `resources.tf` , `variables.tf` , `terraform.tfvars` , and `outputs.tf` .