Refactoring Terraform configuration

At some point in time you might want to refactor (small) parts of the Terraform configuration, e.g.:

- change the name of a resource
- move a resource into a module

This, however, will result in Terraform changing the state

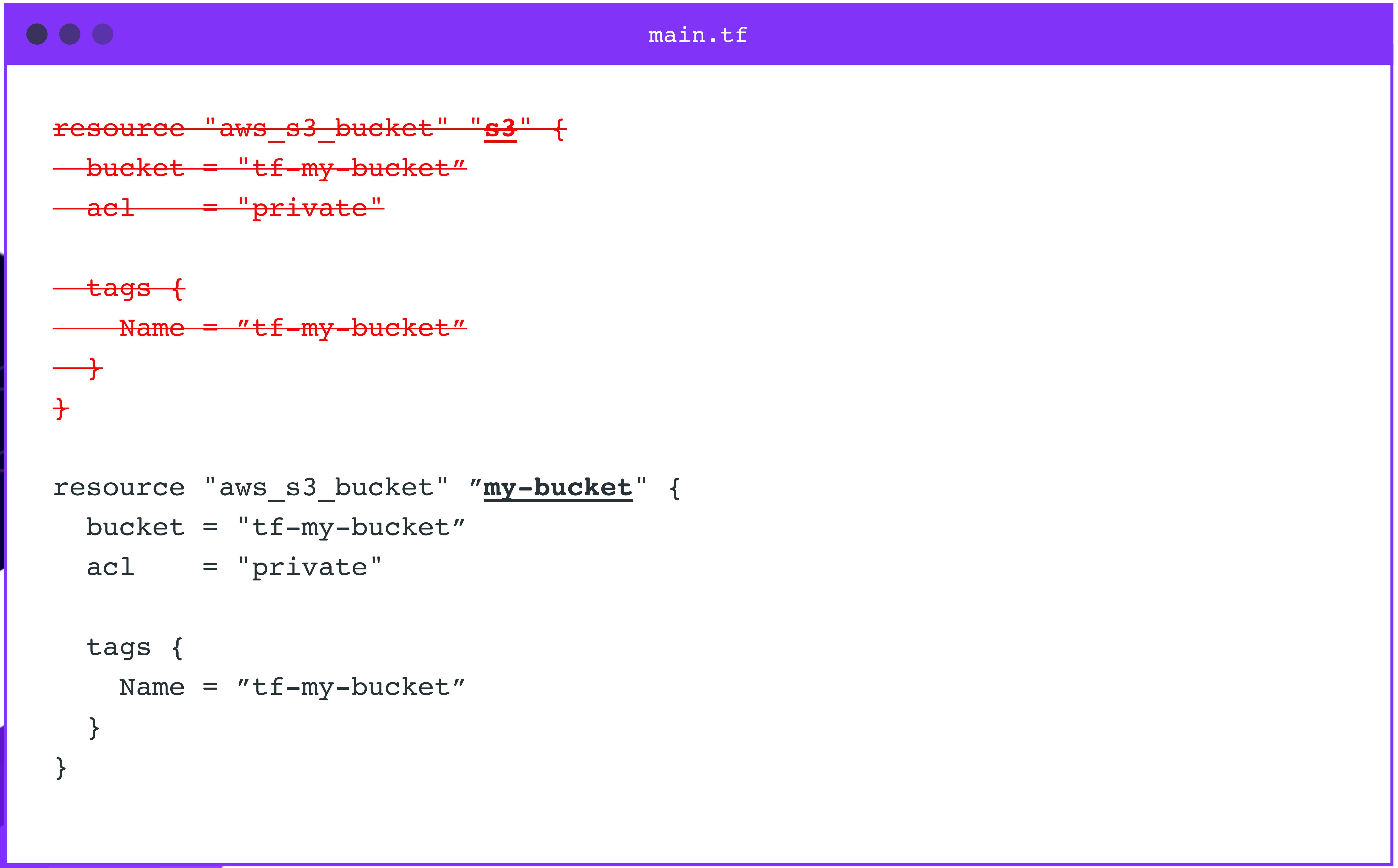
and subsequently changing the existing infrastructure

Command: `terraform state mv`

Move an item matched by the address given to the destination address

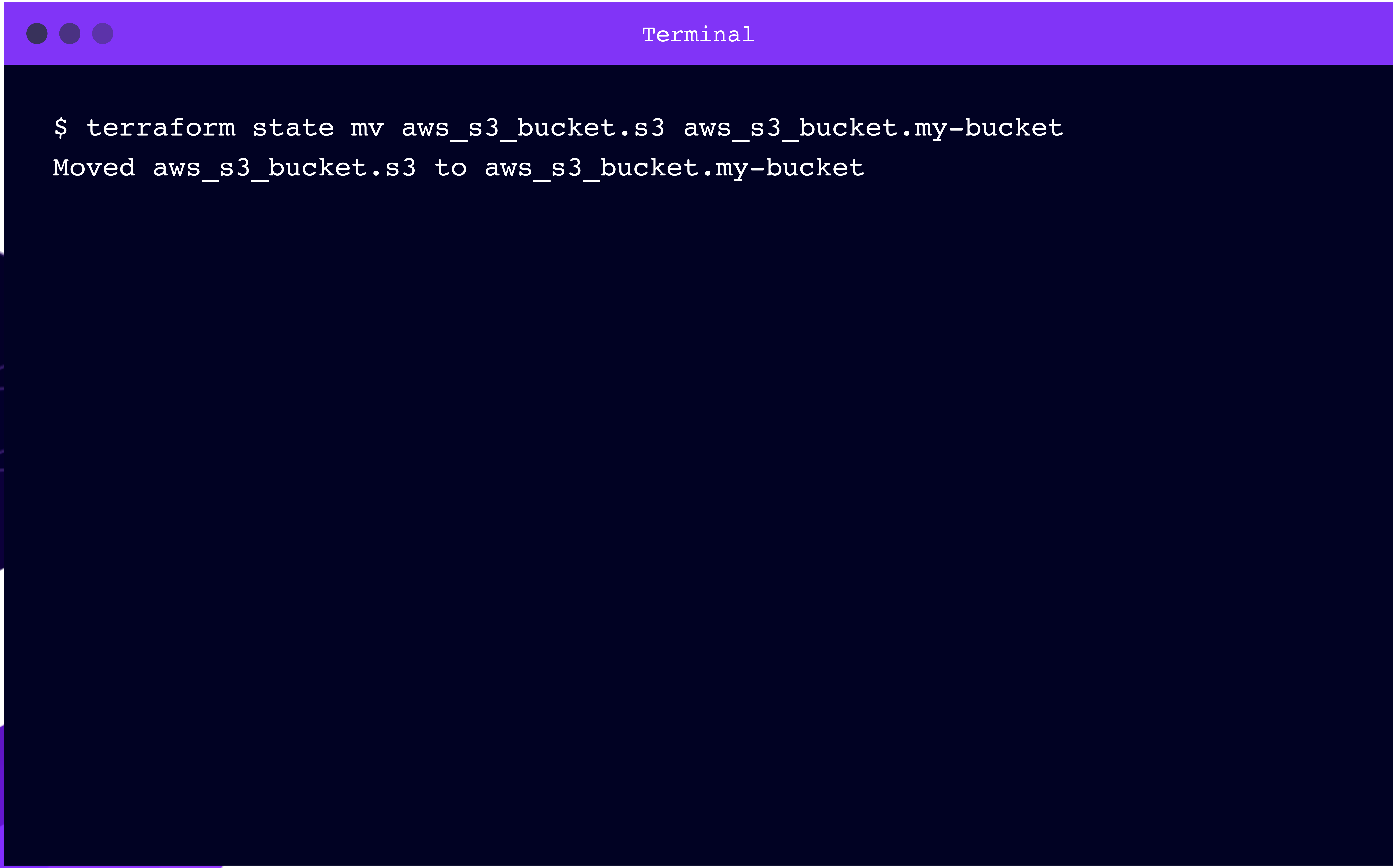
Can move or rename single resources or even entire modules

Can even move resources or modules to a different state file



main.tf

```
~~resource "aws_s3_bucket" "s3" {~~
~~  bucket = "tf-my-bucket”~~
~~  acl    = "private"~~

~~  tags {~~
~~    Name = ”tf-my-bucket”~~
~~  }~~
~~}~~

resource "aws_s3_bucket" ”my-bucket" {
  bucket = "tf-my-bucket”
  acl    = "private"

  tags {
    Name = ”tf-my-bucket”
  }
}
```



```
$ terraform state mv aws_s3_bucket.s3 aws_s3_bucket.my-bucket
Moved aws_s3_bucket.s3 to aws_s3_bucket.my-bucket
```

Specifying Terraform version

Version of Terraform is recorded in state file

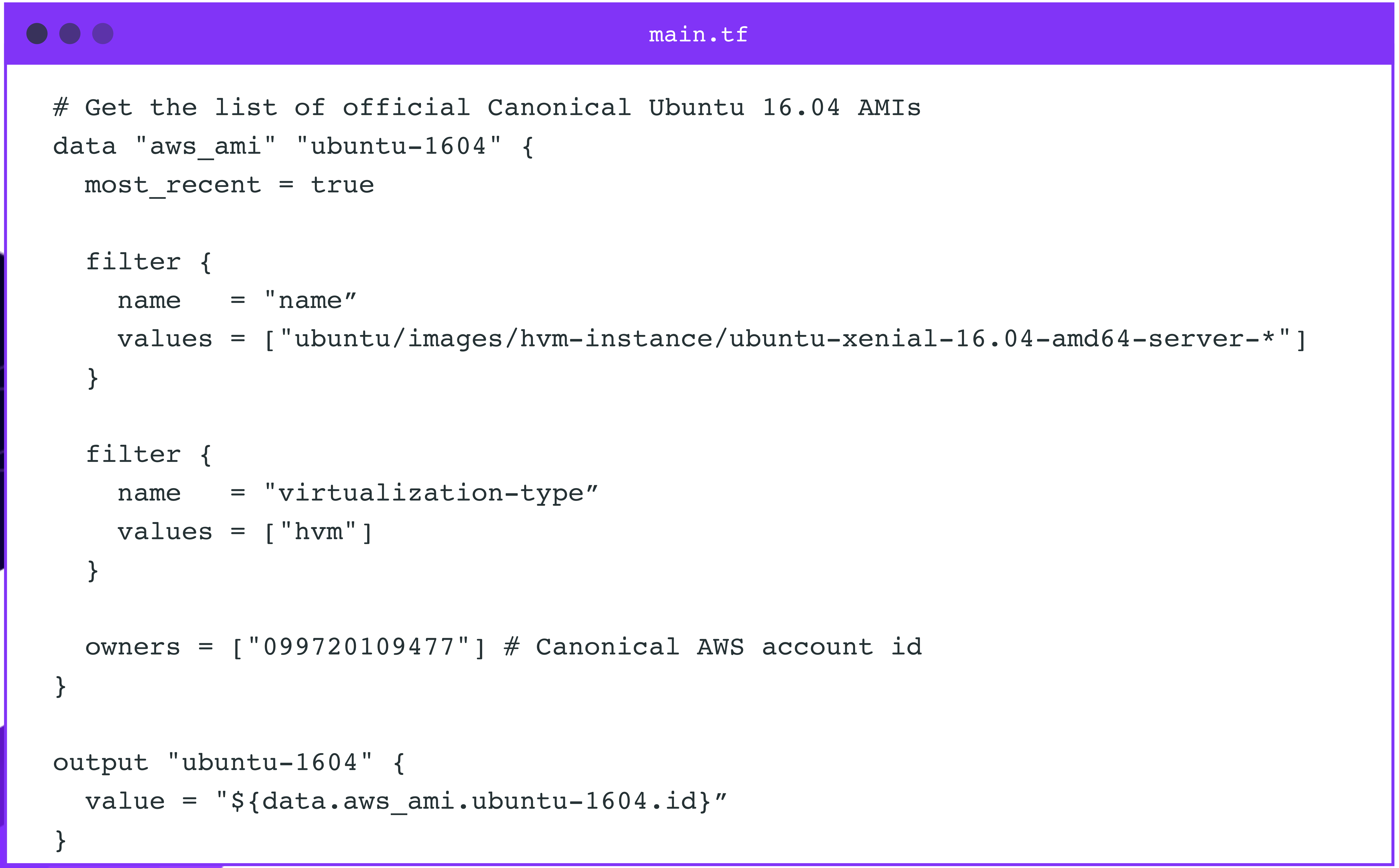
Consequence is that all members within a team need to use the same version of Terraform when working on the same state version.tf

```
terraform {
  required_version = ″= 0.8.8″
}
```

Using Data Sources

Data sources allow data to be fetched or computed for use elsewhere in Terraform configuration

Allows a Terraform configuration to build on information defined outside of Terraform



main.tf

```
# Get the list of official Canonical Ubuntu 16.04 AMIs
data "aws_ami" "ubuntu-1604" {
  most_recent = true

  filter {
    name   = "name”
    values = ["ubuntu/images/hvm-instance/ubuntu-xenial-16.04-amd64-server-*"]
  }

  filter {
    name   = "virtualization-type”
    values = ["hvm"]
  }

  owners = ["099720109477"] # Canonical AWS account id
}

output "ubuntu-1604" {
  value = "${data.aws_ami.ubuntu-1604.id}”
}
```

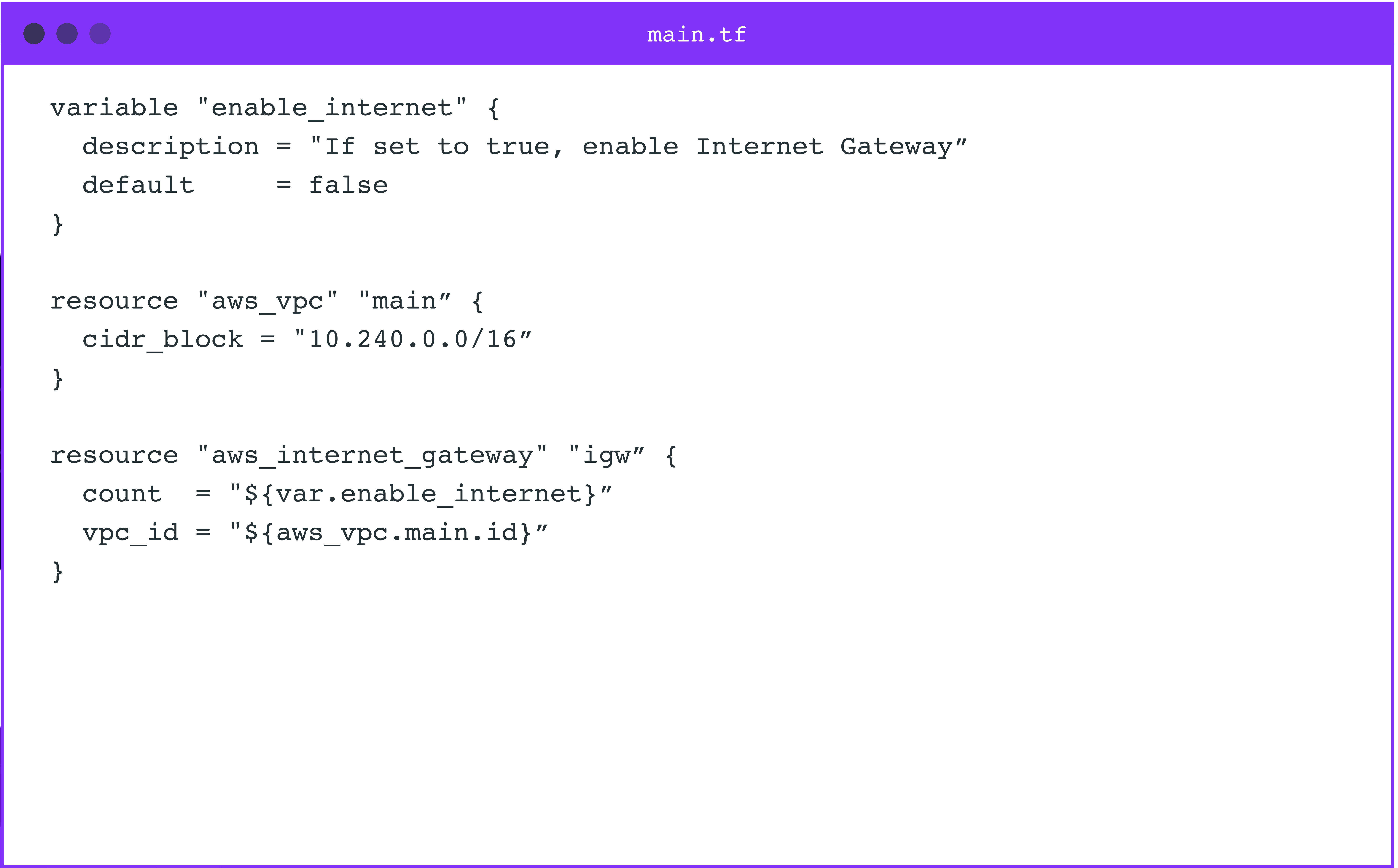
Using Count as an on/off toggle

The Count meta-parameter can be used as a simple on/off toggle

Allowing you to create if-statements in the Terraform configuration

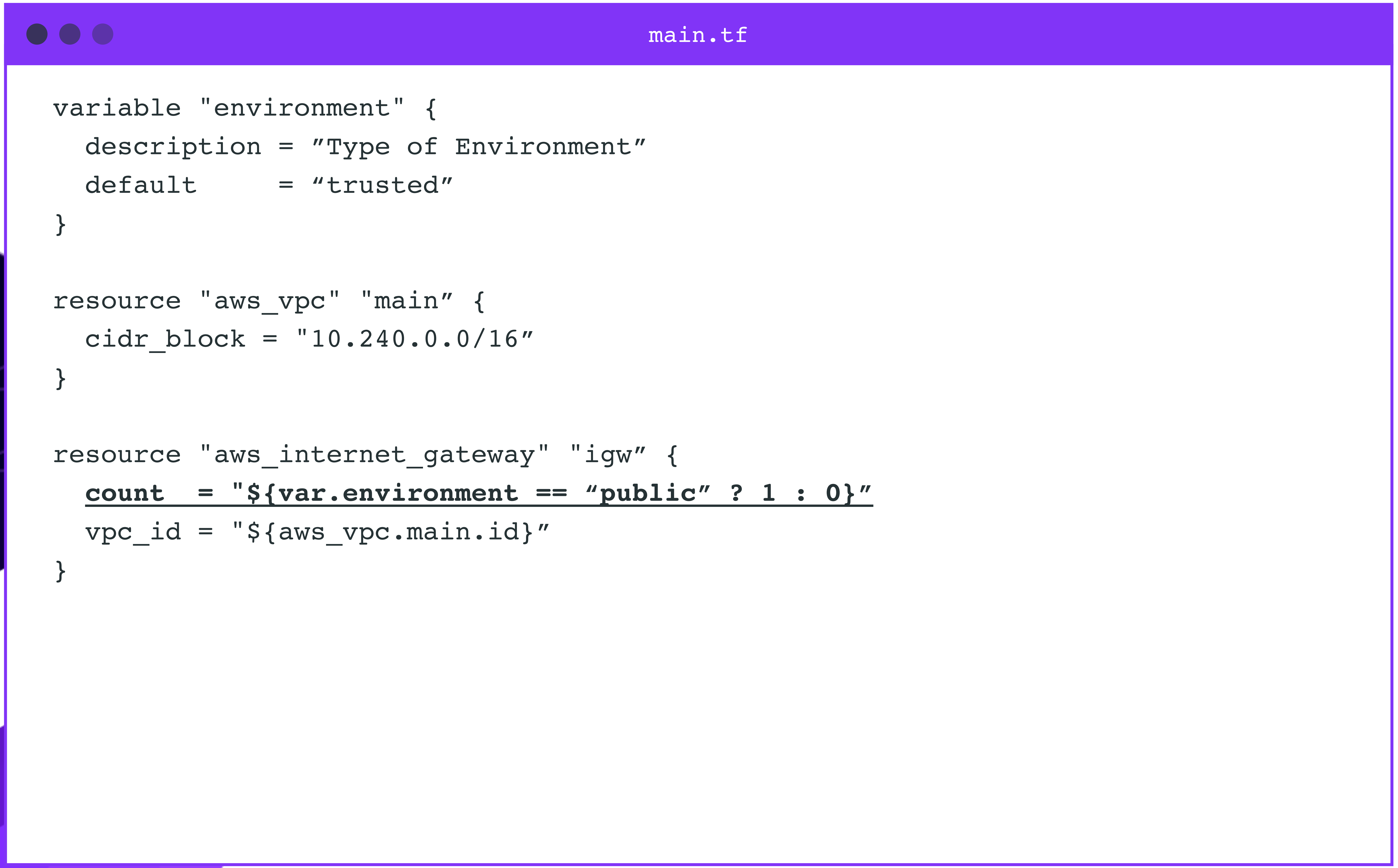
Remember that the `count` argument allows for N number of identical resources

What will happen when the `count` argument is set to 0?



main.tf

```
variable "enable_internet" {
  description = "If set to true, enable Internet Gateway”
  default     = false
}

resource "aws_vpc" "main” {
  cidr_block = "10.240.0.0/16”
}

resource "aws_internet_gateway" "igw” {
  count  = "${var.enable_internet}”
  vpc_id = "${aws_vpc.main.id}”
}
```



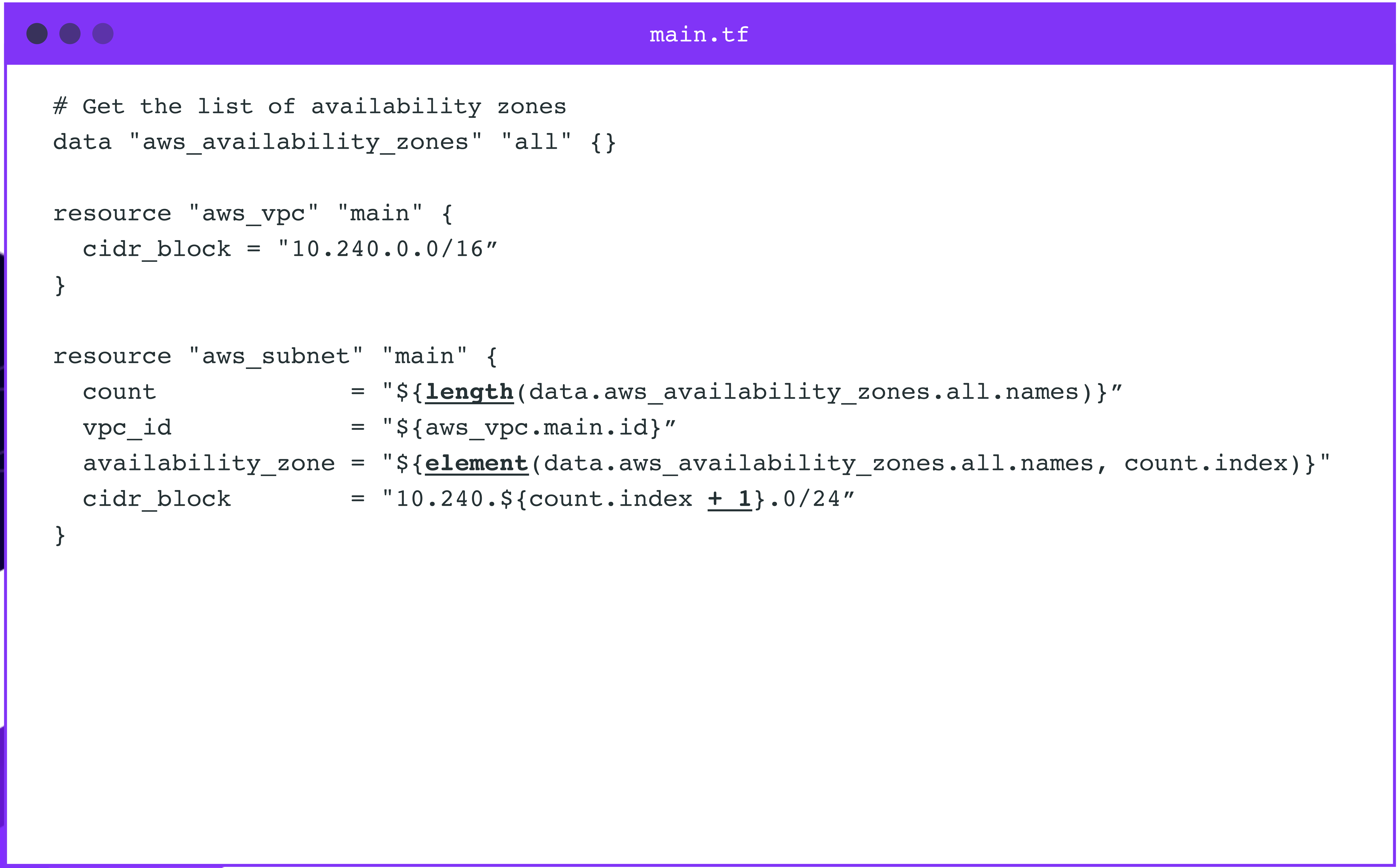
main.tf

```
variable "environment" {
  description = ”Type of Environment”
  default     = “trusted”
}

resource "aws_vpc" "main” {
  cidr_block = "10.240.0.0/16”
}

resource "aws_internet_gateway" "igw” {
  count   = "${var.environment == “public” ? 1 : 0}”
  vpc_id = "${aws_vpc.main.id}”
}
```

Example: Using Data Sources and interpolation

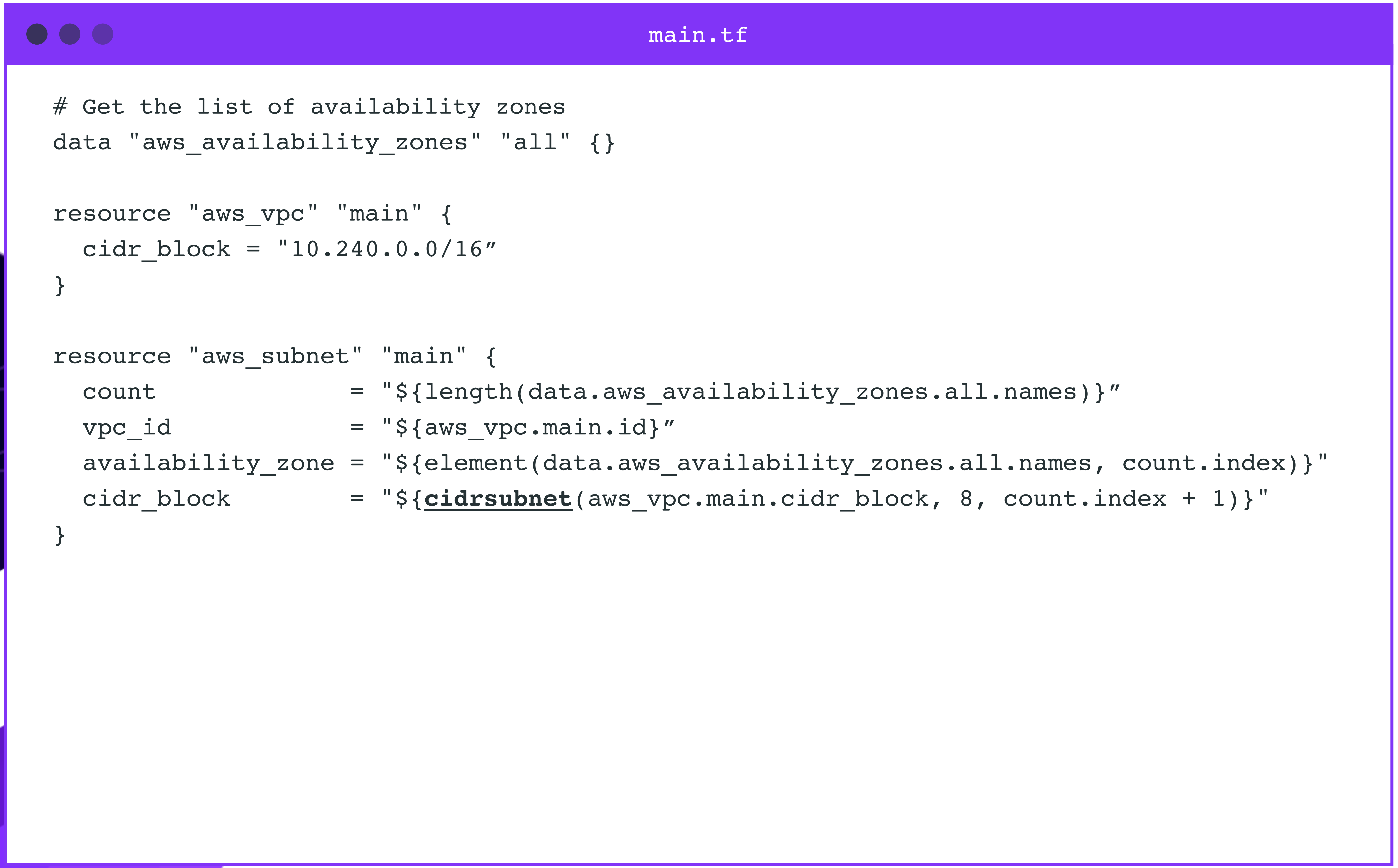
In the next example a Data Source is used to get the available AWS Availability Zones

The Count meta-parameter and some interpolation functions are used to create a subnet for each available AZ



main.tf

```
# Get the list of availability zones
data "aws_availability_zones" "all" {}

resource "aws_vpc" "main" {
  cidr_block = "10.240.0.0/16”
}

resource "aws_subnet" "main" {
  count             = "${length(data.aws_availability_zones.all.names)}”
  vpc_id            = "${aws_vpc.main.id}”
  availability_zone = "${element(data.aws_availability_zones.all.names, count.index)}"
  cidr_block        = "10.240.${count.index + 1}.0/24”
}
```

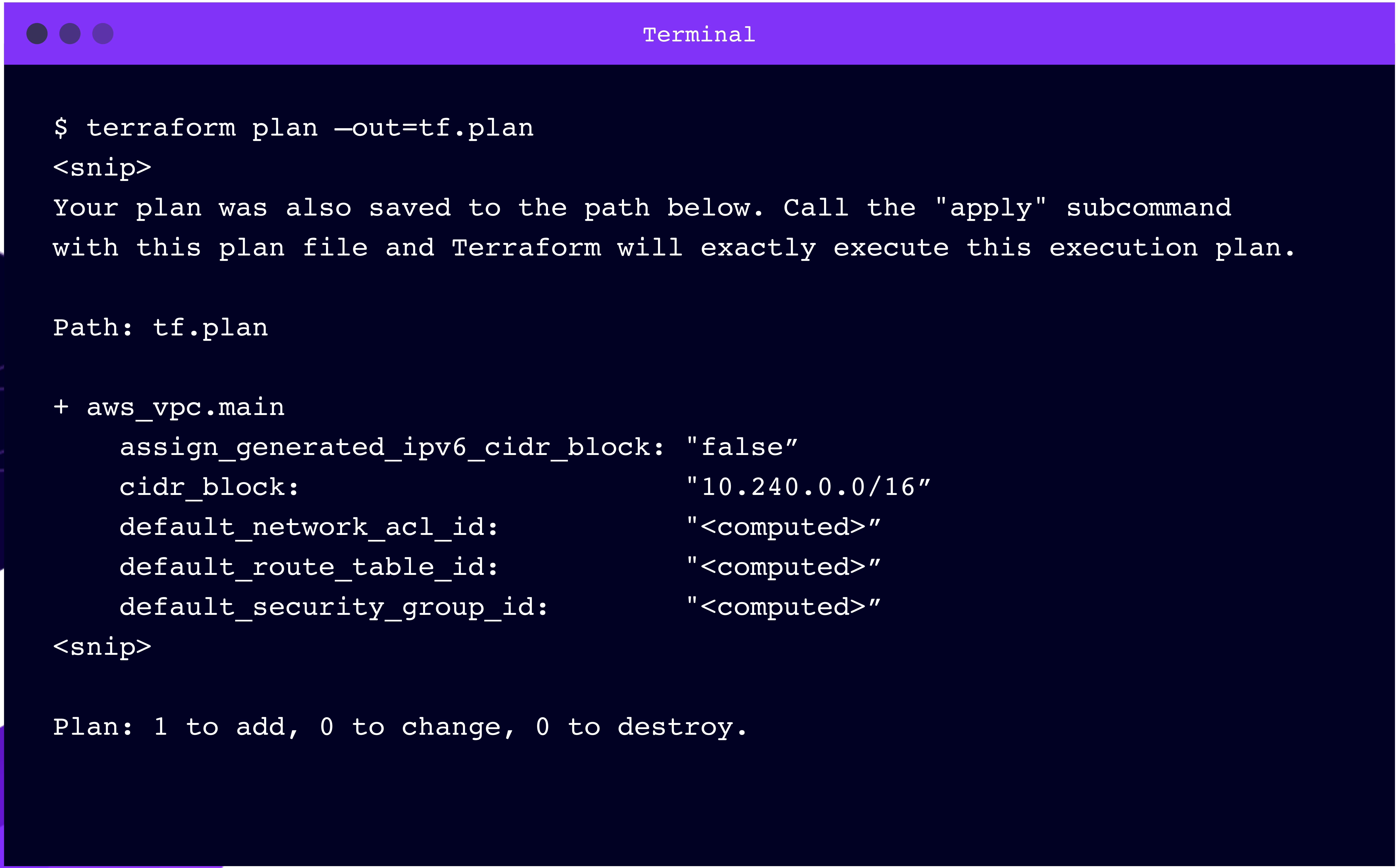


```
# Get the list of availability zones
data "aws_availability_zones" "all" {}

resource "aws_vpc" "main" {
  cidr_block = "10.240.0.0/16”
}

resource "aws_subnet" "main" {
  count             = "${length(data.aws_availability_zones.all.names)}”
  vpc_id            = "${aws_vpc.main.id}”
  availability_zone = "${element(data.aws_availability_zones.all.names, count.index)}"
  cidr_block        = "${cidrsubnet(aws_vpc.main.cidr_block, 8, count.index + 1)}"
}
```

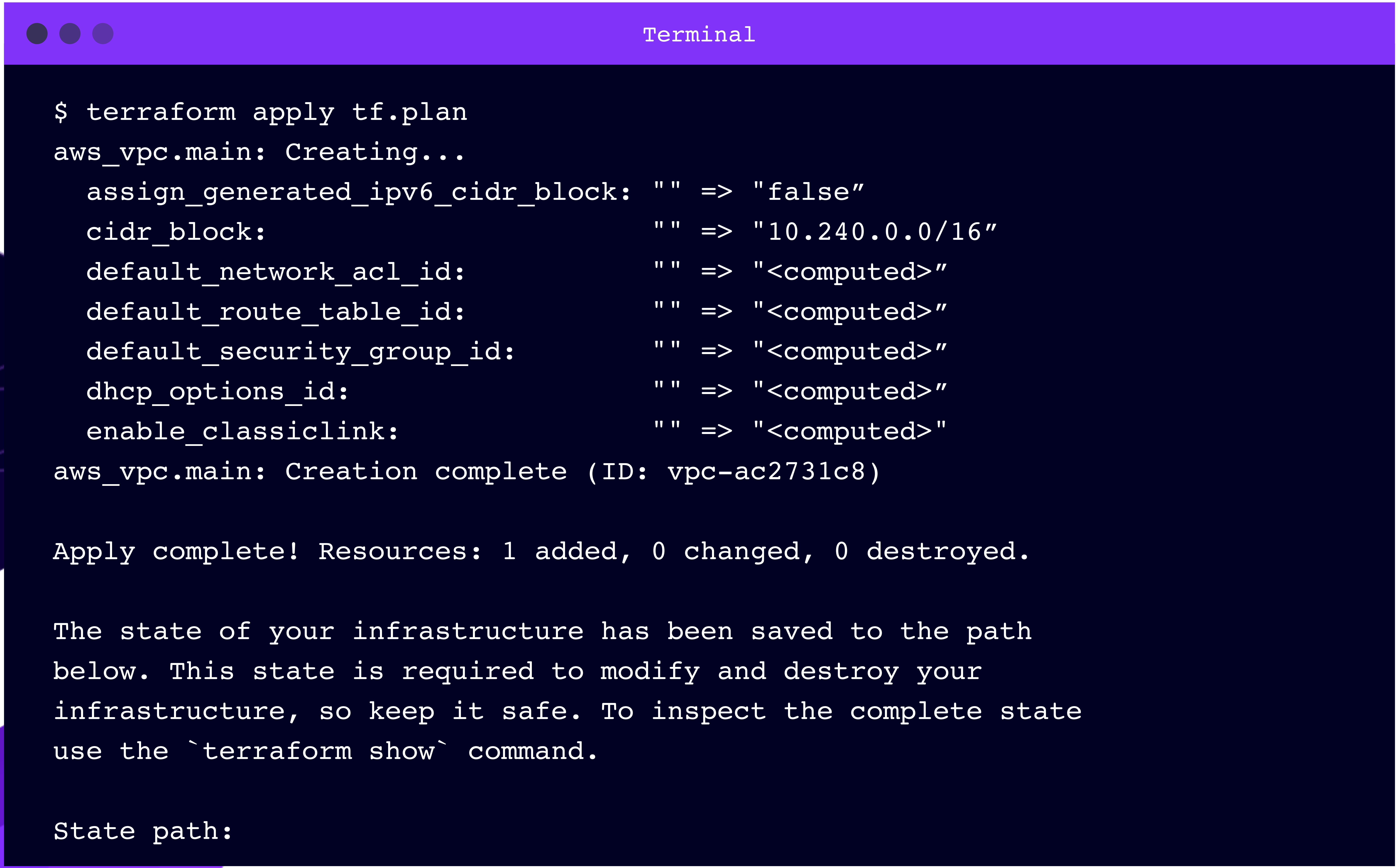
Workflow tip: Run apply only on a plan file

Terraform plan allows you to store the plan in a file.

Terraform apply can use this plan file as an input, to guarantee it executes on this plan only



```
$ terraform plan —out=tf.plan
<snip>
Your plan was also saved to the path below. Call the "apply" subcommand
with this plan file and Terraform will exactly execute this execution plan.

Path: tf.plan

+ aws_vpc.main
    assign_generated_ipv6_cidr_block: "false”
    cidr_block:                       "10.240.0.0/16”
    default_network_acl_id:           "<computed>”
    default_route_table_id:           "<computed>”
    default_security_group_id:        "<computed>”
<snip>

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



```
$ terraform apply tf.plan
aws_vpc.main: Creating...
  assign_generated_ipv6_cidr_block: "" => "false"
  cidr_block:                       "" => "10.240.0.0/16"
  default_network_acl_id:           "" => "<computed>"
  default_route_table_id:           "" => "<computed>"
  default_security_group_id:        "" => "<computed>"
  dhcp_options_id:                  "" => "<computed>"
  enable_classiclink:               "" => "<computed>"
aws_vpc.main: Creation complete (ID: vpc-ac2731c8)

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.

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

State path:
```