JAMES TURNBULL

THE TERRAFORM BOOK

The Terraform Book

James Turnbull

July 15, 2019

Version: v1.5.0 (faad024)

Website: The Terraform Book

ISBN 978-0-9888202-5-8
US$14.99

Contents

Chapter 3

Building an application stack with Terraform

In the last chapter we installed Terraform and got a crash course in the basics of creating, managing, and destroying infrastructure. We learned about Terraform configuration files and the basics of Terraform syntax.

In this chapter we're going to build a more complex infrastructure: a multi-tier web application. We're going to use this exercise to learn more about Terraform configuration syntax and structure.

Our application stack

We're going to build a two-tier web application stack. We're going to build this stack in Amazon Web Services (AWS) in an Amazon VPC environment. We'll create that VPC and the supporting infrastructure as well as the stack itself. The stack will be made up of two components:

- An Amazon Elastic Load Balancer (ELB).
- Two EC2 instances.

The ELB will be load balancing between our two EC2 instances.

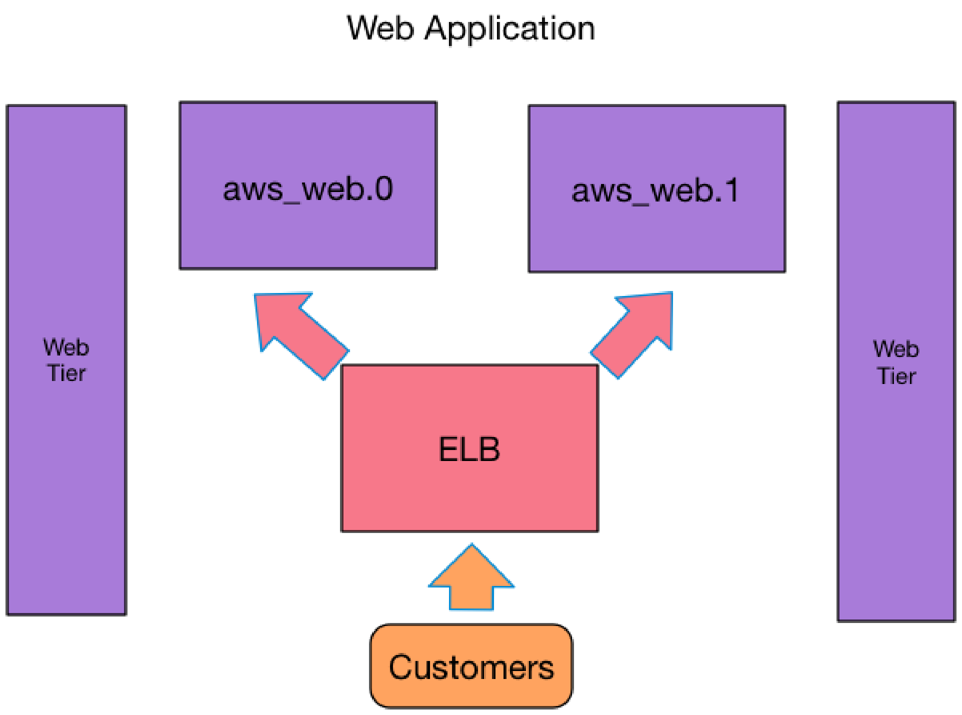


Figure 3.1: Our web application stack

Before we build the stack, we're going to learn about a new concept: parameterizing your configuration.

WARNING If you're following along with this and subsequent chapters, note that we will be creating infrastructure in AWS that will cost small amounts of money to run. We recommend monitoring the infrastructure you're launching in your AWS console and destroying or terminating infrastructure when it is no longer needed.

Parameterizing our configuration

In the previous chapter we created some configuration in our `~/terraform/base/base.tf` configuration file.

Listing 3.1: Our original configuration

```
provider "aws" {
  access_key = "abc123"
  secret_key = "abc123"
  region     = "us-east-1"
}

resource "aws_instance" "base" {
  ami           = "ami-0d729a60"
  instance_type = "t2.micro"
}

resource "aws_eip" "base" {
  instance = aws_instance.base.id
  vpc      = true
}
```

You can see we've hard-coded several attributes in this configuration: the AWS credentials, the AMI, the instance type, and whether our Elastic IP is in a VPC. If we were to expand upon this configuration, we'd end up repeating a number of these values. This is not very DRY. DRY is an abbreviation for "Don't Repeat Yourself," a software principle that recommends reducing the repetition of information. It's also not very practical or efficient if we have to change these values in multiple places.

TIP A little later, in the **Using AWS shared credentials** section, we'll talk more about AWS credentials and how to better protect them.

Variables

In order to address this we're going to parameterize our configuration using variables. Variables allow us to centralize and manage values in our configuration. Let's use the configuration from Chapter 2 to learn more about variables.

We start by creating a file, called `variables.tf`, to hold our variables. We create the file in the `~/terraform/base` directory.

Listing 3.2: Creating the variables.tf file

```
$ cd ~/terraform/base
$ touch variables.tf
```

TIP The file can be called anything. We've just named it `variables.tf` for convenience and identification. Remember all files that end in `.tf` will be loaded by Terraform.

Let's create a few variables in this file now. Variables can come in a number of types, for example:

- Strings — String syntax. Can also be Boolean's: `true` or `false`.
- Maps — An associative array or hash-style syntax.
- Lists — An array syntax.

Let's take a look at some string variables first.

Listing 3.3: Our first variables

```
variable "access_key" {
  type = string
  description = "The AWS access key."
}

variable "secret_key" {
  type = string
  description = "The AWS secret key."
}

variable "region" {
  type = string
  description = "The AWS region."
  default     = "us-east-1"
}
```

Terraform variables are created with a `variable` block. They have a name and optional type, default, and description arguments.

TIP You can learn more about variables in the Terraform variable documentation.

Types

We've specified the variable type for all our variables. Variable types tell Terraform the type of value of the variable, like `string`, `number`, or `bool`. Terraform uses types and type constraints to ensure the data used in variables and resources is correct, failing early rather than parsing bad data.

You can also specify collections of types, like lists or maps. You can create collection variables by specifying the type of collection and the type of values contained in the collection, for example a list of strings.

Listing 3.4: Variable collection type specified

```
variable "region_list" {
  type = list(string)
  description = "AWS availability zones."
  default = ["us-east-1a", "us-east-1b"]
}
```

This would create a list of strings.

TIP You can also specify any as the type to allow the collection to contain any type of value.

If you omit the `type` attribute then Terraform assumes your variable is a string, unless the `default` is in the format of another variable type. Here Terraform would assume the first variable is a string but that the second is a list of strings.

Listing 3.5: Variable type specified

```
variable "region" {
  description = "The AWS region."
  default = "us-east-1"
}

variable "region_list" {
  description = "AWS availability zones."
  default = ["us-east-1a", "us-east-1b"]
}
```

TIP You can learn more about types and type constraints in the Terraform type documentation.

Descriptions

You can also supply an optional description of the variable using the `description` attribute.

Listing 3.6: Variable descriptions

```
variable "region" {
  description = "The AWS region."
  default = "us-east-1"
}
```

TIP We recommend you always add variable types and descriptions. You never know who'll be using your code, and it'll make their (and your) life a lot easier if every variable has a clear type and description. Comments are fun too.

Values

There are a number of ways to populate the value of a variable, which we'll see shortly. The value of our first two variables is currently undefined. The third variable, the `region`, has a default value set with the `default` attribute. This is useful if you want the variable set, even if it's not populated by any other mechanism.

Using variables

Let's update our `provider` with the new variables we've just created.

Listing 3.7: Adding our new variables

```
provider "aws" {
  access_key = var.access_key
  secret_key = var.secret_key
  region     = var.region
}
```

Each variable is identified as a variable by the `var.` prefix. Currently only one of these variables has a value, the `default` of `us-east-1` we've set for the `var.region` variable. Soon we'll see how to populate values for the other variables but let's first explore some of the variable types we'll be using.

TIP Since Terraform 0.8 there is a command called `terraform console`. The console is a Terraform REPL that allows you to work with your resources. It's a good way to explore working with Terraform syntax. You can read about it in the console command documentation.

Maps

Most of our variable examples have, thus far, been strings. We can also specify other types of variables, for example maps and lists. Let's look at maps first.

NOTE See a full list of variables in the variables documentation.

Maps are associative arrays and ideal for situations where you want to use one value to look up another value. For example, one of our potential configuration attributes is the EC2 instance's AMI. AMIs are region specific, so if we change region we will need to look up a new AMI. Terraform's maps are ideal for this task.

Let's define a map of strings in our `variables.tf` file.

Listing 3.8: A map variable

```
variable "ami" {
  type = map(string)
  default = {
    us-east-1 = "ami-0d729a60"
    us-west-1 = "ami-7c4b331c"
  }
  description = "The AMIs to use."
}
```

We can see our new variable is called `ami`. We've specified a `type` of `map(string)`. The `(string)` after the `map` indicates this is a map of strings. We've also specified default values for two keys: the `us-east-1` and `us-west-1` regions.

So how do we use this map variable? Let's update `base.tf` with the `ami` variable.

Listing 3.9: Using map variables in base.tf

```
provider "aws" {
  access_key = var.access_key
  secret_key = var.secret_key
  region     = var.region
}

resource "aws_instance" "base" {
  ami           = var.ami[var.region]
  instance_type = "t2.micro"
}

resource "aws_eip" "base" {
  instance = aws_instance.base.id
  vpc      = true
}
```

You can see we've used two variables to populate our `ami` argument: `var.ami` and `var.region`, like so:

```
var.ami[var.region]
```

This performs a lookup of the `var.ami` variable using the value of the `var.region` variable. So if our `var.region` variable was set to `us-west-1`, then our `ami` attribute would receive a value of `ami-7c4b331c`.

You can also look up maps explicitly, for example, `var.ami["us-west-1"]` will get the value of the `us-west-1` key from the `var.ami` map variable.

Finally, Terraform also has a set of built-in functions to make it easier to work with variables and values. This includes functions to work with collections. For maps, we have a function available called `lookup` that lookups up the value of a map like so:

```
lookup(map, key)
```

Or in the case of our lookup:

```
lookup(var.ami, var.region)
```

TIP You can find a full list of functions in the Terraform functions reference.

Lists

Another useful variable type available in Terraform is the list. Let's assume we have a list of security groups strings we'd like to add to our instances. Our list would be constructed like so:

Listing 3.10: Constructing a list

```
variable "security_group_ids" {
  type    = list(string)
  description = "List of security group IDs."
  default = ["sg-4f713c35", "sg-4f713c35", "sg-4f713c35"]
}
```

We can specify a list directly as the value of a variety of attributes, for example:

Listing 3.11: Using a list

```
resource "aws_instance" "base" {

. . .

  vpc_security_group_ids = var.security_group_ids
}
```

NOTE You'll need to create some security groups if you want to test this and use the resulting IDs in your list.

Lists are zero-indexed. We can retrieve a single element of a list using the syntax:

`var.variable[element]`

Like so:

Listing 3.12: Retrieving a list element

```
resource "aws_instance" "base" {

. . .

  vpc_security_group_ids = var.security_group_ids[1]
}
```

This will populate the `vpc_security_group_ids` attribute with the second element in our `var.security_group_ids` variable.

You can also use a function called `element` to retrieve a value from a list.

`element(list, index)`

Or for the second element again:

`element(var.security_group_ids, 1)`

TIP Again, you can find a full list of functions in the Terraform functions reference.

Variable defaults

Variables with and without defaults behave differently. A defined, but empty, variable is a required value for an execution plan.

Listing 3.13: An empty variable

```
variable "access_key" {
  type = string
  description = "The AWS access key."
}
```

If you run a Terraform execution plan then it will prompt you for the value of `access_key` (and any other empty variables).

Let's try that now.

Listing 3.14: Empty and default variables

```
$ terraform plan
var.access_key
  Enter a value: abc123

var.secret_key
  Enter a value: abc123

. . .
```

We can see that Terraform has prompted us to provide values for two variables: `var.access_key` and `var.secret_key`. Again, the `var` prefix indicates this is a variable, and the suffix is the variable name. Setting the variables for the plan will not persist them. If you re-run `terraform plan`, you'll again be prompted to set values for these values.

So how does Terraform populate and persist variables?

Populating variables

Of course, inputting the variable values every time you plan or apply Terraform configuration is not practical. To address this, Terraform has a variety of methods by which you can populate variables. Those ways, in order of descending resolution, are:

1. Loading variables from command line flags.
2. Loading variables from a file.
3. Loading variables from environment variables.
4. Variable defaults.

Loading variables from command line flags

The first method allows you to pass in variable values when you run `terraform` commands.

Listing 3.15: Command line variables

```
$ terraform plan -var 'access_key=abc123' -var 'secret_key=
abc123'
```

We can also populate maps via the `-var` command line flag:

Listing 3.16: Setting a map with var

```
$ terraform plan -var 'ami={ us-east-1 = "ami-0d729a60", us-west-
1 = "ami-7c4b331c" }'
```

And lists via the command line:

Listing 3.17: Populating a list via command line flag

```
$ terraform plan -var 'security_group_ids=["sg-4f713c35", "sg-4
f713c35", "sg-4f713c35"]'
```

You can pass these variables on both the `plan` and `apply` commands. Obviously, like the input prompt, this does not persist the values of variables. Next time you run Terraform, you'll again need to specify these variables values.

Loading variables from a file

Our next method, populating variable values via files, does allow persistence. When Terraform runs it will look for a file called `terraform.tfvars`. We can populate this file with variable values that will be loaded when Terraform runs.

Let's create that file now.

Listing 3.18: Creating a variable assignment file

```
$ touch terraform.tfvars
```

We can then populate this file with variables—here a string, map, and list respectively.

Listing 3.19: Adding variable assignments

```
access_key = "abc123"
secret_key = "abc123"
ami = {
  us-east-1 = "ami-0d729a60"
  us-west-1 = "ami-7c4b331c"
}
security_group_ids = [
  "sg-4f713c35",
  "sg-4f713c35",
  "sg-4f713c35"
]
```

When Terraform runs it will automatically load the `terraform.tfvars` file and assign any variable values in it. The file can contain Terraform configuration syntax or JSON, just like normal Terraform configuration files.

Any variable for which you define a value needs to exist. In our case, the variables `access_key`, `secret_key`, and `security_group_ids` need to be defined with `variable` blocks in our `variables.tf` file. If they do not exist you'll get an error like so:

Listing 3.20: Variable doesn't exist error

```
module root: 1 error(s) occurred:

* provider config 'aws': unknown variable referenced: '
access_key'. define it with 'variable' blocks
```

You can also name the `terraform.tfvars` file something else—for example, we could have a variable file named `base.tfvars`. If you do specify a new file name, you will need to tell Terraform where the file is with the `-var-file` command line flag.

Listing 3.21: Running Terraform with a custom variable file

```
$ terraform plan -var-file base.tfvars
```

TIP You can use more than one `-var-file` flag to specify more than one file. If you specify more than one file, the files are evaluated from first to last, in the order specified on the command line. If a variable value is specified multiple times, the last value defined is used.

Loading variables from environment variables

Terraform will also parse any environment variables that are prefixed with `TF_VAR`. For example, if Terraform finds an environment variable named:

`TF_VAR_access_code=abc123`

it will use the value of the environment variable as the string value of the `access_code` variable.

We can populate a map via an environment variable:

`TF_VAR_ami='{us-east-1 = "ami-0d729a60", us-west-1 = "ami-7c4b331c"}'`

and a list.

`TF_VAR_roles='["sg-4f713c35", "sg-4f713c35", "sg-4f713c35"]'`

TIP Variable files and environment variables are a good way of protecting passwords and secrets. This avoids storing them in our configuration files, where they might end up in version control. A better way is obviously some sort of secrets store. Since Terraform 0.8 there is now support for integration with Vault for secrets management.

Variable defaults

Lastly, you can specify variable defaults for your variables.

Listing 3.22: Variable defaults

```
variable "region" {
  type = string
  description = "The AWS region."
  default = "us-east-1"
}
```

Variable defaults are specified with the `default` attribute. If nothing in the above list of variable population methods resolves the variable then Terraform will use the default.

TIP Terraform also has an "override" file construct. When Terraform loads configuration files it appends them. With an override the files are instead merged. This allows you to override resources and variables.

Our new variables are useful syntax. Let's start our build using some of this new syntax.

Starting our stack

Now that we've learned how to parameterize our configuration, let's get started with building a new application stack. Inside our `~/terraform` directory let's create a new directory called `web` to hold our stack configuration, and let's initialize it as a Git repository.

Listing 3.23: Creating the web directory

```
$ cd ~/terraform
$ mkdir web
$ cd web
$ git init
```

Let's add our `.gitignore` file too. We'll exclude any state files to ensure we don't commit any potentially sensitive variable values.

Listing 3.24: Adding the state file and backup to .gitignore

```
$ echo "terraform.tfstate*" >> .gitignore
$ git add .gitignore
$ git commit -m "Adding .gitignore file"
```

It's important to note that this is a new configuration. Terraform configurations in individual directories are isolated. Our new configuration in the `web` directory will, by default, not be able to refer to, or indeed know about, any of the configuration in the `base` directory. We'll see how to deal with this in Chapter 5, when we talk more about state.

NOTE You can find the code for this chapter on GitHub.

Let's create a new file to hold our stack configuration, a file to define our variables, and a file to populate our variables.

Listing 3.25: Creating the stack files

```
$ touch web.tf variables.tf terraform.tfvars
```

Let's begin by populating our `variables.tf` file.

Listing 3.26: Our variables.tf file

```
variable "access_key" {
  type = string
  description = "The AWS access key."
}
variable "secret_key" {
  type = string
  description = "The AWS secret key."
}
variable "region" {
  type = string
  description = "The AWS region."
}
variable "key_name" {
  type = string
  description = "The AWS key pair to use for resources."
}
variable "ami" {
  type = map(string)
  type    = "map"
  description = "A map of AMIs."
  default = {}
}
variable "instance_type" {
  type = string
  description = "The instance type."
  default = "t2.micro"
}
```

Note that we've used variables similar to our example in Chapter 2. We've also added a few new variables, including the name of a key pair we're going to use for our instances, and a map that will specify the AMI we wish to use.

TIP You should have created a key pair when you set up AWS.

Let's populate some of these variables by adding definitions to our `terraform.tfvars` file.

Listing 3.27: The web terraform.tfvars file

```
access_key = "abc123"
secret_key = "abc123"
region = "us-east-1"
ami = {
  us-east-1 = "ami-f652979b"
  us-west-1 = "ami-7c4b331c"
}
```

You can see we've provided values for our AWS credentials, the region, and a map of AMIs for the `us-east-1` and `us-west-1` regions.

Using AWS shared credentials

We mentioned earlier that we don't have to specify our credentials in the `terraform.tfvars` file. Indeed, it's often a very poor security model to specify these credentials in a file that could easily be accidentally distributed or added to version control. Instead of specifying the credentials in your configuration, you should configure the AWS client tools. These provide a shared credential configuration that Terraform can consume, removing the need to specify credentials.

To install the AWS client tools on Linux, we'd use Python `pip`:

Listing 3.28: Installing AWS CLI on Linux

```
$ sudo pip install awscli
```

On OS X we can use `pip` or `brew` to install the AWS CLI:

Listing 3.29: Installing AWS CLI on OSX

```
$ brew install awscli
```

On Windows, we'd use the MSI installer from AWS, or if you've used the Chocolatey package manager, we'd install via the `choco` binary:

Listing 3.30: Installing awscli via choco

```
C:\> choco install awscli
```

We then run the `aws` binary with the `configure` option.

Listing 3.31: Running aws configure

```
$ aws configure
AWS Access Key ID [None]: abc123
AWS Secret Access Key [None]: abc123
Default region name [None]: us-east-1
Default output format [None]:
```

You would replace each `abc123` with your AWS credentials and specify your preferred default region. This will create a file in `~/.aws/credentials` with your credentials that will look like:

Listing 3.32: The aws/credentials file

```
[default]
aws_access_key_id = abc123
aws_secret_access_key = abc123
```

And a file called `~/.aws/config`, with our default region:

Listing 3.33: The aws/config file

```
[default]
region = us-east-1
```

TIP Due to a bug with Terraform, you will still need to specify `region = us-east-1` (or your region) in your Terraform configurations. This is because Terraform does not seem to read the `config` file in some circumstances.

Now we can remove the `var.access_key` and `var.secret_key` variables from our `variables.tf` and `terraform.tfvars` files if we wish.

For the rest of the book we'll assume you have configured shared credentials, and we'll remove references to the access and secret keys!

For the other variables in our `variables.tf` file, we're going to rely on their defaults.

TIP We could also use the `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` environment variables to specify our credentials. Terraform also automatically consumes these variables. If you're on OS X, you should also look at envchain, which uses the OS X Keychain to help manage environment variables.

First resources

Now that we've got the inputs for our stack defined, let's start to create our resources and their configuration in the `web.tf` file.

Listing 3.34: Our web.tf file

```
provider "aws" {
  region = var.region
}

module "vpc_basic" {
  source        = "./vpc"
  name          = "web"
  cidr          = "10.0.0.0/16"
  public_subnet = "10.0.1.0/24"
}

resource "aws_instance" "web" {
  ami                         = var.ami[var.region]
  instance_type               = var.instance_type
  key_name                    = var.key_name
  subnet_id                   = module.vpc_basic.
public_subnet_id
  associate_public_ip_address = true
  user_data                   = file("files/web_bootstrap.sh")

  vpc_security_group_ids = [
    "aws_security_group.web_host_sg.id",
  ]

  count = 2
}

resource "aws_elb" "web" {
  name            = "web-elb"
  subnets         = module.vpc_basic.public_subnet_id
  security_groups = aws_security_group.web_inbound_sg.id
  listener {
    instance_port     = 80
. . .
  }
  instances       = aws_instance.web.*.id
}

resource "aws_security_group" "web_inbound_sg" {
. . .
```

TIP You'll find code you can download and use for this example on GitHub.

You can see we've first added the aws provider to allow us to provision our resources from AWS. We've omitted the access and secret access keys from our provider because we've assumed we're using our AWS shared configuration to provide them. The only option we have specified for the provider is the region.

As we discussed in Chapter 2, you can define multiple providers, both for different services and for different configurations of service. A common Terraform pattern is to define multiple providers aliased for specific attributes—for example, being able to create resources in different AWS regions. Here's an example:

Listing 3.35: Multiple providers

```
provider "aws" {
  region = var.region
}

provider "aws" {
  alias  = "west"
  region = "us-west-2"
}

resource "aws_instance" "web" {
  provider = "aws.west"
. . .
}
```

We've defined an aws provider that uses our var.region variable to define the AWS region to which we'll connect. We've then defined a second aws provider with an alias attribute of west and the region hard-coded to us-west-2.

We can now refer to this specific provider by using the `provider` attribute. The `provider` is a special type of attribute called a meta-argument. Meta-arguments are attributes you can add to any resources in Terraform. Terraform has a number of meta-arguments available, and we'll see others later in the book.

TIP The depends_on attribute we mentioned in the last chapter is also a meta-argument.

We're going to stick with our single `aws` provider for now and use a single AWS region.

You can also see that we've added some new configuration syntax and structures to our `web.tf` file. Let's look at each of these now, starting with the `module` structure.

Modules

Modules are defined with the `module` block. Modules are a way of constructing reusable bundles of resources. They allow you to organize collections of Terraform code that you can share across configurations.

Often you have a configuration construct such as infrastructure like an AWS VPC, an application stack, or other collection of resources that you need to repeat multiple times in your configurations. Rather than cutting and pasting and repeating all the resources required to configure that infrastructure, you can bundle them into a module. You can then reuse that module in your configurations.

You can configure inputs and outputs for modules: an API interface to your modules. This allows you to customize them for specific requirements, while your code remains as DRY and reusable as possible.

TIP Hashicorp makes available a collection of verified and community modules in the Terraform Module Registry. These include modules for a large number of purposes and are a good point to start if you need a module. You can learn more about the Terraform Module Registry in the documentation.

Defining a module

To Terraform, every directory containing configuration is automatically a module. Using modules just means referencing that configuration explicitly. References to modules are created with the `module` block.

Listing 3.36: The vpc_basic module

```
module "vpc_basic" {
  source = "./vpc_basic"
  name   = "web"
  cidr   = "10.0.0.0/16"
  public_subnet = "10.0.1.0/24"
}
```

As you can see, modules look just like resources only without a type. Each module requires a name. The module name must be unique in the configuration.

Modules only have one required attribute: the module's `source`. The `source` tells Terraform where to find the module's source code. You can store modules locally in your filesystem or remotely in repositories such as GitHub. In our case the `vpc_basic` module is located in a directory called `vpc_basic` inside our `~/terraform/web` directory.

You can specify a module multiple times in a configuration by giving it a new name but specifying the same source. For example:

Listing 3.37: Multiple vpc_basic modules

```
module "vpc_basic_a" {
  source = "./vpc_basic"
. . .
}

module "vpc_basic_b" {
  source = "./vpc_basic"
. . .
}
```

Here Terraform would create two VPCs, one from `vpc_basic_a` and the other from `vpc_basic_b`. We would configure each differently.

Let's create the `vpc_basic` directory first and initialize it as a Git repository, because ultimately we want to store our module on GitHub.

Listing 3.38: Creating the vpc_basic module directory

```
$ pwd
~/terraform/web
$ mkdir vpc_basic
$ cd vpc_basic
$ git init
```

Inside our `source` attribute we specify the `vpc_basic` directory relative to the `~/terraform/web` directory. Remember Terraform uses the current directory it's in when executed as its root directory. To ensure Terraform finds our module we need to specify the `vpc` directory relative to the current directory.

TIP This path manipulation in Terraform is often tricky. To help with this, Terraform provides a built-in variable called `path`. You can read about how to use the `path` variable in the interpolation path variable documentation.

Instead of storing them locally, you can also specify remote locations for your modules. For example:

Listing 3.39: The vpc_basic module with a remote source

```
module "vpc_basic" {
  source = "github.com/turnbullpress/tf_vpc_basic
. . .
}
```

This will load our module from a GitHub repository:

`https://github.com/turnbullpress/tf_vpc_basic`

This allows us to reference module configurations without needing to store them in directories underneath or adjacent to our configuration.

This also allows us to create versioning for modules. Terraform can refer to a specific repository branch or tag as the source of a module. For example,

Listing 3.40: Referencing a module version

```
module "vpc_basic" {
  source = "github.com/turnbullpress/tf_vpc_basic.git?ref=
production"
}
```

The `ref=` suffix can be a branch name, a tag, or a commit. Here we're downloading the `production` branch of the module in the `tf_vpc_basic` repository.

Or if you want to get a module specifically from the Terraform Registry then you can use syntax like so:

Listing 3.41: Referencing a registry module

```
module "vpc" {
  source = "terraform-aws-modules/vpc/aws"
}
```

The source path format for Terraform Registry modules looks like this:

`namespace/name/provider`

The `namespace` is like an organization or source of the module. The `name` is the module's name and the `provider` is the specific provider it uses. The module's homepage will contain full documentation on how to use it, including any required inputs and any outputs.

NOTE Modules with a blue tick on the Terraform Registry are verified and from a Hashicorp partner. These modules should be more resilient and tested than others. You can also publish your own modules on the Registry.

Terraform Registry modules can also be versioned and you can use a specific version of a module like so:

Listing 3.42: Referencing a registry module's version

```
module "vpc" {
  source = "terraform-aws-modules/vpc/aws"
  version = "1.3.0"
}
```

TIP

You can find a full list of the potential sources and how to configure them in the module source documentation.

Module structure

Inside our `vpc_basic` directory our module is identical to any other Terraform configuration. It will have variables, variable definitions, and resources.

Variables

Let's start with creating a file to hold the module's variables. We'll use a file called `interface.tf`.

TIP The explicit file name makes it clear that this is the module's API, the interface to the module.

Listing 3.43: Creating the vpc_basic module variables

```
$ cd vpc_basic
$ touch interface.tf
```

We populate this file with the variables we'll use to configure the VPC that the module is going to build.

Listing 3.44: The vpc_basic module's variables

```
variable "name" {
  type        = string
  description = "The name of the VPC."
}
variable "cidr" {
  type        = string
  description = "The CIDR of the VPC."
}
variable "public_subnet" {
  type        = string
  description = "The public subnet to create."
}
variable "enable_dns_hostnames" {
  type        = bool
  description = "Should be true if you want to use private DNS
within the VPC"
  default     = true
}
variable "enable_dns_support" {
  type        = bool
  description = "Should be true if you want to use private DNS
within the VPC"
  default     = true
}
```

You can see that we've defined a number of variables. Some of the variables will be required: `name`, `cidr`, and `public_subnet`. These variables currently have no defaults, so we must specify a value for each of them. We've specified the values in the `module` block in our `web.tf` file. This represents the incoming API for the `vpc_basic` module.

Listing 3.45: The vpc_basic module's default variables

```
module "vpc_basic" {
  source = "./vpc_basic"
  name   = "web"
  cidr   = "10.0.0.0/16"
  public_subnet = "10.0.1.0/24"
}
```

If we do not specify a required variable, then Terraform will fail with an error:

```
Error loading Terraform: module root: module vpc_basic: required
variable cidr not set
```

NOTE So what's `module root`? Well, remember that modules are just folders containing files. Terraform considers every folder of configuration files a module. Terraform has created an implicit module, called the `root` module, from the stack configuration contained in the `~/terraform/web` directory.

We also have several variables with defaults that we can override when configuring our module.

Listing 3.46: Overriding vpc_basic module's default variables

```
module "vpc_basic" {
  source = "./vpc_basic"
. . .
  enable_dns_hostnames = false
}
```

This will override the default value of the `enable_dns_hostnames` variable and set it to `false`.

Module resources

Now let's add the resources to configure our VPC. We'll create a configuration file called `main.tf` to hold the resources and then populate it.

Listing 3.47: The vpc_basic module resources

```
resource "aws_vpc" "tfb" {
  cidr_block           = var.cidr
  enable_dns_hostnames = var.enable_dns_hostnames
  enable_dns_support   = var.enable_dns_support
  tags = {
    Name = var.name
  }
}

resource "aws_internet_gateway" "tfb" {
  vpc_id = aws_vpc.tfb.id
  tags = {
    Name = "${var.name}-igw"
  }
}

resource "aws_route" "internet_access" {
  route_table_id         = aws_vpc.tfb.main_route_table_id
  destination_cidr_block = "0.0.0.0/0"
  gateway_id             = aws_internet_gateway.tfb.id
}

resource "aws_subnet" "public" {
  vpc_id                  = aws_vpc.tfb.id
  cidr_block              = var.public_subnet
  map_public_ip_on_launch = var.map_public_ip_on_launch
  tags = {
    Name = "${var.name}-public"
  }
}
```

TIP Our VPC module is very simple. It does not expose anywhere near the complexity of a complete VPC configuration. For a more fully featured module take a look at the Terraform Community VPC module.

You can see we've added a number of new resources but what you can't see is an `aws` provider definition. This is because we don't need one. The module will, by default, inherit the provider configuration from the `web.tf` file and use that to connect to AWS.

Module provider inheritance

However if you have multiple providers specified in the `web.tf` file, as we saw earlier using the `alias` attribute, then you must explicitly tell the module which provider to use!

TIP This change occurred in Terraform 0.11 and later.

So if our `web.tf` file has defined two providers:

Listing 3.48: Multiple aliased providers

```
provider "aws" {
  alias  = "use1"
  region = "us-east-1"
}

provider "aws" {
  alias  = "uws2"
  region = "us-west-2"
}
```

One aliased `use1` and one aliased `usw2` then you must explicitly tell the module which provider to use. We do this using the `providers` meta-argument.

Listing 3.49: The vpc_basic module's default variables

```
module "vpc_basic" {
  source = "./vpc_basic"
  name   = "web"
  cidr   = "10.0.0.0/16"
  public_subnet = "10.0.1.0/24"
  providers = {
    "aws" = "aws.use1"
  }
}
```

Here we've specified a new attribute for our module: `providers`. The `providers` attribute contains a list of each of the providers our module uses and the alias name of the specific provider to use. So in this case for our `aws` provider the `vpc` module will use the `aws.use1` definition of the provider.

TIP You can read more about module provider inheritance in the modules documentation.

Our module resources

So back to our module's resources. You can see that we've used a series of new AWS resource types: the VPC itself, gateways, routes, and subnets. Let's look at the `aws_vpc` resource in more detail.

Listing 3.50: The aws_vpc resource

```
resource "aws_vpc" "tfb" {
  cidr_block           = var.cidr
  enable_dns_hostnames = var.enable_dns_hostnames
  enable_dns_support   = var.enable_dns_support
  tags = {
    Name = var.name
  }
}
```

We've called our VPC resource `aws_vpc.tfb`.[1] Inside the resource, we've passed some of our variables in—for example, the `var.cidr` variable to configure the resource. We need to ensure each of these variables is defined and that they're either populated in the `module` block or that a default exists in the module for them.

In the `main.tf` file we also configure a series of other resources using a mix of variables and resource references as the values of our attributes—for example, in the `aws_subnet` resource:

Listing 3.51: The aws_subnet resource

```
resource "aws_subnet" "public" {
  vpc_id                  = aws_vpc.tfb.id
  cidr_block              = var.public_subnet
  tags = {
    Name = "${var.name}-public"
  }
}
```

Our resource is named `aws_subnet.public` and references attributes from earlier

[1] tfb for The Terraform Book

configured resources. For example, the `vpc_id` attribute is populated from the ID of the `aws_vpc.tfb` resource we created earlier in the module.

```
aws_vpc.tfb.id
```

Another interesting attribute is the `Name` tag we've created. Here we've used the interpolated `var.name` variable inside a string.

```
${var.name}-public
```

This will create a value that combines the value of the `var.name` variable with the string `-public`.

The combination of these resources will create an Amazon VPC with access to the Internet, internal routing, and a single public subnet, specified in CIDR notation.

Outputs

Lastly, we need to specify outputs from our module. This is essentially the API response from using the module. They can contain useful data like the IDs of resources created or other configuration that we might want to use outside of the module to configure other resources. To add these outputs we use a new construct called an `output`.

The `output` construct can be used in any Terraform configuration, not just in modules. It is a way to highlight specific information from the attributes of resources we're creating. This allows us to selectively return critical information to the user or to another application rather than returning all the possible attributes of all resources and having to filter the information down.

Let's add some outputs to the end of our `interface.tf` file.

Listing 3.52: The vpc_basic module outputs

```
output "public_subnet_id" {
  value = aws_subnet.public.id
}

output "vpc_id" {
  value = aws_vpc.tfb.id
}

output "cidr" {
  value = aws_vpc.tfb.cidr_block
}
```

Here one of our outputs is the VPC ID. We've called the output `vpc_id`. The output will return the `aws_vpc.tfb.id` attribute value from the `aws_vpc.tfb` resource we created inside the module.

You can see that, like a variable, an `output` is configured as a block with a name. Each `output` has a value, usually an interpolated attribute from a resource being configured.

TIP Since Terraform 0.8, you can also add a `description` attribute to your outputs, much like you can for your variables.

Outputs can also be marked as containing sensitive material by setting the `sensitive` attribute.

Listing 3.53: The vpc_basic module outputs

```
output "public_subnet_id" {
  value     = aws_subnet.public.id
  sensitive = true
}
```

When outputs are displayed—for instance, at the end of the application of a plan—sensitive outputs are redacted, with `<sensitive>` displayed instead of their value.

NOTE This is purely a visual change. The outputs are not encrypted or protected. This is more to keep the information out of logs, for example a build system.

We'll see how to use these outputs inside our stack configuration shortly.

NOTE We recommend using a naming convention for Terraform files inside modules. This isn't required but it makes code organization and comprehension easier. We use `interface.tf` for variables and outputs and `main.tf` for resources.

With that our module is complete. Now let's see it at work.

Using our module

Back in our `web.tf` configuration file we've already defined our `module` block.

Listing 3.54: The vpc_basic module block revisited

```
module "vpc_basic" {
  source = "./vpc_basic"
  name   = "web"
  cidr   = "10.0.0.0/16"
  public_subnet = "10.0.1.0/24"
}
```

Inside the `module` block we've passed in all of our required variables and when Terraform runs it will:

1. Load the module code.
2. Pass in the variables.
3. Create all the resources in the module.
4. Return the outputs.

Let's see how we can use those outputs in our stack's configuration.

We have the `aws_elb.web` or AWS Elastic Load Balancer resource. To configure it we need to provide at least one piece of information from our `vpc` module: the subnet ID of the subnet to which our load balancer is connected.

Listing 3.55: Our web aws_elb resource

```
resource "aws_elb" "web" {
  name            = "web-elb"
  subnets         = [module.vpc_basic.public_subnet_id]
  security_groups = [aws_security_group.web_inbound_sg.id]
  listener {
    instance_port     = 80
    instance_protocol = "http"
    lb_port           = 80
    lb_protocol       = "http"
  }
  instances       = aws_instance.web.*.id
}
```

We've specified a module output for the value of the `subnets` attribute. We've wrapped it in `[ ]` to convert it into a list because the `subnets` argument expects a list value to be returned, even though in our case we're only returning a single value. This converts the string returned by the module to a list.

```
[module.vpc_basic.public_subnet_id]
```

TIP You can read more about type conversion in the Terraform expressions documentation.

This variable is an output from our `vpc` module. Module outputs are prefixed with `module`, the module name—here `vpc_basic`—and then the name of the output. You can access any output you've defined in the module.

It's important to remember that a module's resources are isolated. You only see the data you define. You must specify outputs for any attribute values you want to expose from them.

Like resources, modules automatically create dependencies and relationships. For example, by using the `module.vpc_basic.public_subnet_id` output from the `vpc_basic` module we've created a dependency relationship between the `aws_elb.web` resource and the `vpc_basic` module.

TIP Since Terraform 0.8, you can also specify the `depends_on` meta-argument to explicitly create a dependency on a module. You can reference a module via name, for example `module.vpc_basic`.

We can use this combination of variables and outputs as a simple API for our modules. It allows us to define standard configuration in the form of modules and then use the outputs of those modules to ensure standardization of our resources.

TIP The fine folks at Segment.io have released an excellent tool called `terraform-docs`. The terraform-docs tool reads modules and produces Markdown or JSON documentation for the module based on its variables and outputs.

Getting our module

Before you can use a module in your configuration, you need to load it or `get` it. You do that from the `~/terraform/web` directory, via the `terraform get` command.

Listing 3.56: The Terraform get command

```
$ pwd
~/terraform/web
$ terraform get
Get: file:///Users/james/terraform/web/vpc_basic
```

This gets the module code and stores it in the `.terraform/modules` directory inside the `~/terraform/web` directory.

If you change your module, or the module you're using has been updated, you'll need to run the `get` command again, with the `-update` flag set.

Listing 3.57: Updating a module

```
$ terraform get -update
```

If you run the `terraform get` command without the `-update` flag, Terraform will not update the module.

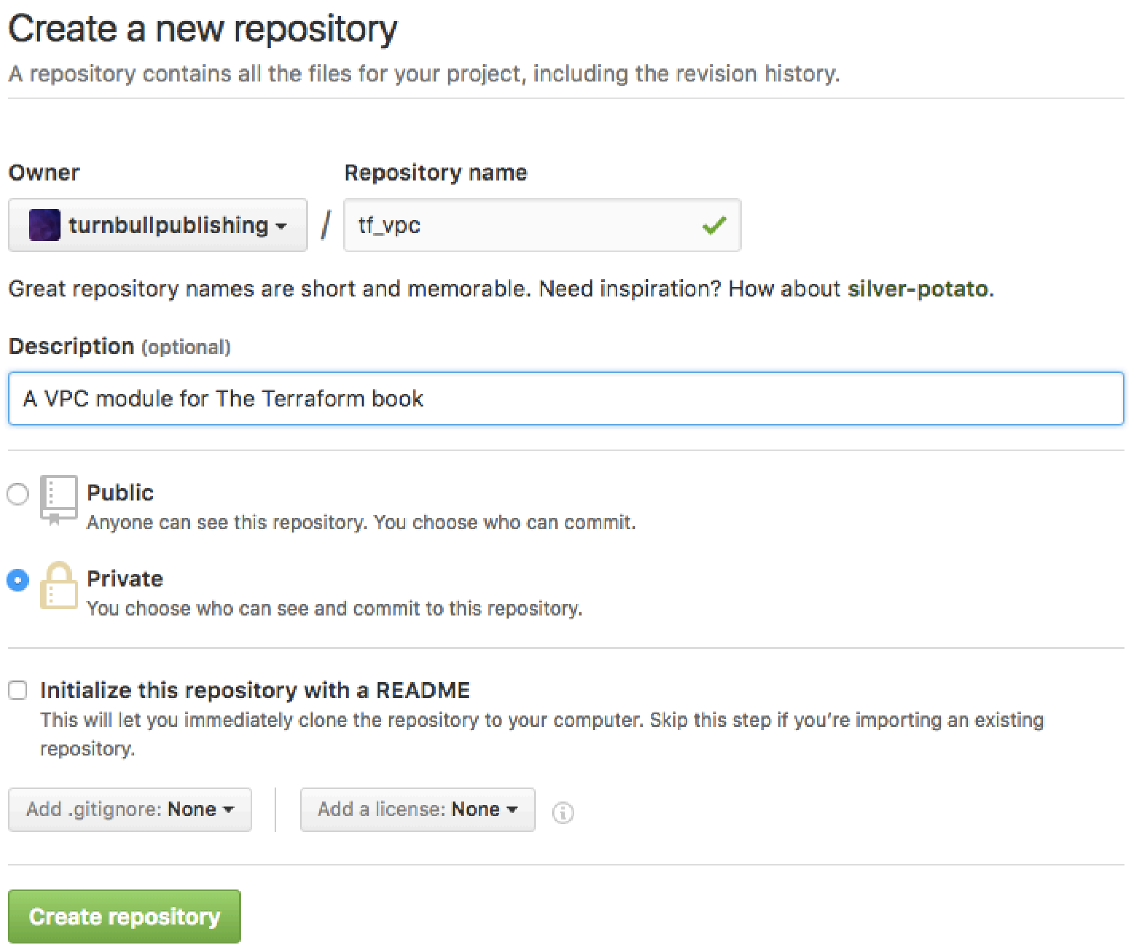
Moving our module to a repository

Currently our `vpc_basic` module is located in our local filesystem. That's cumbersome if we want to reuse it. Let's instead move it to a GitHub repository.

You'll need a GitHub account to do this. You can join GitHub on their site. There's also some useful sign-up documentation available.

After we've created our GitHub account, we can create a new GitHub repository. We're calling ours `turnbullpress/tf_vpc_basic`.

NOTE You'd use your own GitHub username and repository name.



Create a new repository

A repository contains all the files for your project, including the revision history.

Owner: turnbullpublishing / Repository name: tf_vpc ✓

Great repository names are short and memorable. Need inspiration? How about **silver-potato**.

Description (optional): A VPC module for The Terraform book

○ Public — Anyone can see this repository. You choose who can commit.

[✓ Private] — You choose who can see and commit to this repository.

☐ Initialize this repository with a README — This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Figure 3.2: Creating a GitHub repository

Let's add a `README.md` file to our `~/terraform/web/vpc_basic` directory to tell folks how to use our module.

Listing 3.58: The README.md file

```
# AWS VPC module for Terraform

A lightweight VPC module for Terraform.

## Usage

module "vpc_basic" {
  source = "github.com/turnbullpress/tf_vpc_basic"
  name   = "vpc_name"
  cidr   = "10.0.0.0/16"
  public_subnet = "10.0.1.0/24"
}

See `interface.tf` for additional configurable variables.

## License

MIT
```

Let's create a `.gitignore` file to ensure we don't accidentally commit any state or variables values we don't want in our module repository.

Listing 3.59: Creating a .gitignore file

```
$ echo ".terraform/" >> .gitignore
$ echo "terraform.tfvars" >> .gitignore
$ git add .gitignore
```

We can then commit and push our `vpc` module.

Listing 3.60: Committing and pushing our vpc_basic module

```
$ pwd
~/terraform/web/vpc_basic
$ git add .
$ git commit -m "First commit of vpc_basic module"
$ git tag -a "v0.0.1" -m "First release of vpc_basic module"
$ git remote add origin git@github.com:turnbullpress/
tf_vpc_basic.git
$ git push -u origin master --tags
```

Here we've added all the `vpc_basic` module files and committed them. We've also tagged that commit as `v0.0.1`. We add the newly created remote repository and push up our code and tag.

Now we can update our `module` configuration in `web.tf` to reflect the new location of the `vpc` module.

Listing 3.61: Updating our vpc_basic module configuration

```
module "vpc_basic" {
  source = "github.com/turnbullpress/tf_vpc_basic.git?ref=v0
.0.1"
  name    = "web"
  cidr    = "10.0.0.0/16"
  public_subnet = "10.0.1.0/24"
}
```

We'll need to get our module again since we've changed its source.

Listing 3.62: Getting the new vpc_basic module

```
$ terraform get
Get: git::https://github.com/turnbullpress/tf_vpc_basic.git?ref=
v0.0.1
```

Any time we want to use the `vpc_basic` module, we can now just reference the module on GitHub. This also means we can manage multiple versions of the module—for example, we could create `v0.0.2` of the module, and then use the `ref` parameter to refer to that.

```
git::https://github.com/turnbullpress/tf_vpc_basic.git?ref=v0.0.2
```

This allows us to test a new version of a module without changing the old one.

Counts and counting

Let's go back to our `web.tf` file and look at our remaining resources. We know we want to create two EC2 instances in our stack. We know we can only specify a resource named `aws_instances.web` once. It doesn't make sense to duplicate the resource with a new name, especially if its configuration is otherwise identical.

In a traditional programming language this is when you'd break out a `for` loop. Terraform has two solutions for this: counts and the for loop introduced in Terraform 0.12. In this chapter, we're going to focus on counts. A count is another meta-argument and can be added to any resource.

TIP Terraform has a number of meta-arguments available.

You add a count to a resource to have Terraform iterate and create the number of resources equal to the value of the count. Let's look at how count works with our aws_instances.web resource.

Listing 3.63: The aws_instances count

```
resource "aws_instance" "web" {
  ami           = var.ami[var.region]
  instance_type = var.instance_type
  key_name      = var.key_name
  subnet_id     = module.vpc_basic.public_subnet_id
  associate_public_ip_address = true
  user_data     = file("files/web_bootstrap.sh")
  vpc_security_group_ids = [
    aws_security_group.web_host_sg.id,
  ]
  count         = 2
}
```

We've added the count meta-argument and specified a value of 2. When Terraform creates the aws_instances.web resource it will iterate and create two of these resources. It'll create each resource with the index of the count suffixed to the resource name, like so:

- aws_instance.web[0]
- aws_instance.web[1]

We can now refer to these resources and their attributes using these names. For example, to access the id of one of these instances we'd use:

aws_instance.web[0].id

Sets of counted resources using splat

Sometimes we want to refer to the set of resources created via a `count`. To do this Terraform has a splat syntax: `*`. This allows us to refer to all of these resources in a variable. Let's see how that works in the `aws_elb.web` resource.

Listing 3.64: The aws_elb resource

```
resource "aws_elb" "web" {
  name            = "web-elb"
  subnets         = [module.vpc_basic.public_subnet_id]
  security_groups = [aws_security_group.web_inbound_sg.id]

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

  instances = aws_instance.web.[*].id
}
```

The `instances` attribute in our `aws_elb.web` resource needs to contain a list of the IDs of all the EC2 instances that are connected to our load balancer. To provide this we make use of the splat syntax like so:

`aws_instance.web.[*].id`

The value assigned to the attribute is a list interpolated from the IDs of all our EC2 instances.

Setting values with count indexes

We can also use the `count` to allow us to specify different values for an attribute for each iteration of a resource. We do this by referring to the index of a count in a variable.

Let's take a look at how this works. We start by declaring a list variable with a value for each iteration.

Listing 3.65: Using count indexes in variables.tf

```
variable "instance_ips" {
  description = "The IPs to use for our instances"
  default = ["10.0.1.20", "10.0.1.21" ]
}
```

We've defined a new list variable called `instance_ips` that contains two IP addresses in our VPC subnet. We're going to match the index of the count with the relevant element of the list.

Listing 3.66: Looking up the count index

```
resource "aws_instance" "web" {

. . .

  private_ip    = var.instance_ips[count.index]

. . .

  count = length(var.instance_ips)
}
```

You can see we've updated our `aws_instance.web` resource to add the `private_ip` attribute. The value of the attribute uses the list element lookup we saw earlier in this chapter.

For the element lookup we specify the name of the variable we just defined: `var.instance_ips` and the index of the count using `count.index`. The `count.index` is a special function on the `count` meta-argument to return the index.

When each `aws_instance.web` resource is created, the matching list element will be retrieved with the index from the `count.index`. The `var.instance_ips` will return each value, and the instance will get the correct IP address, hence:

- `aws_instance.web[0]` will get the IP address `10.0.1.20`.
- `aws_instance.web[1]` will get the IP address `10.0.1.21`.

This makes it easier to customize individual resources in a collection.

You'll notice we also changed the value of the `count` meta-argument. Instead of hard-coding a number, we used the length of the `var.instance_ips` list as the value of the `count`. We know the `var.instance_ips` list needs to have an IP address for each instance otherwise instance creation will fail. So we know that we can only have as many instances as the number of elements in this list. The length function allows us to count the element in this list and return an integer, in our case `2`. We can use this to populate the `count` attribute. This means we can increment the number of instances created by just adding new private IP addresses, rather than having to change and track the instance count in two places.

Listing 3.67: Using the length function

```
resource "aws_instance" "web" {

. . .

  count = length(var.instance_ips)
}
```

We can also use the `count.index` in other places. For example, to add a unique name to our EC2 instances we could do the following:

Listing 3.68: Naming using the count.index

```
resource "aws_instance" "web" {
. . .

  tags = {
    Name = "web-${format("%03d", count.index)}"
  }
  count  = length(var.instance_ips)
}
```

This will populate the `Name` tag of each instance with a name based on the `count.index`. We've also used a new function called `format`. The `format` function formats strings according to a specified format. Here we're turning the `count.index` of `0` or `1` into a three-digit number.

The `format` function is essentially a `sprintf` and is a wrapper around Go's `fmt` library syntax. So `%03d` is constructed from `0`, indicating that you want to pad the number to the specified width with leading zeros. Then `3` indicates the width that you want, and `d` specifies a base 10 integer. The flags together will pad single digits with a leading `0` but ignore numbers larger than three digits in length.

This will produce `Name` tags `web-000` and `web-001` respectively. Having a `web-000` is a bit odd though. It comes from `count`'s zero index. Alternately we can use some math in our interpolated string like so:

Listing 3.69: Interpolated math

```
  tags = {
    Name = "web-${format("%03d", count.index + 1)}"
  }
```

This would add one to every `count.index` value producing the tags `web-001` and `web-002` respectively. We can do other math: subtract, multiple, divide, etc., on any integer or float variables.

We can also iterate through list elements with `count.index`. Let's create a list variable with some tags we'd like to add to our instances.

Listing 3.70: AWS owner tags in variables.tf

```
variable "owner_tag" {
  default = ["team1", "team2"]
}
```

We'd like to distribute our instances between these two tag values in `web.tf`.

Listing 3.71: Splitting up the count instances

```
resource "aws_instance" "web" {
. . .

  tags = {
    Owner = var.owner_tag[count.index]
  }
  count   = length(var.instance_ips)
}
```

This returns the element matching the `count.index` from the specified list variable. When we create the resources, one instance will be tagged `team1` and the second `team2`.

If we specify more instances than the number of elements in our list, then Terraform will fail with an error like:

Listing 3.72: Count exhausted failure

```
* index 2 out of range for list var.owner_tag (max 2) in:

var.owner_tag[count.index]
```

Wrapping counts with the element function

We can, however, cause Terraform to wrap the list using the `element` function. The `element` function pulls an element from a list using the given index and wraps when it reaches the end of the list.

Let's update our code to do that.

Listing 3.73: Wrapping the count instances list

```
resource "aws_instance" "web" {
. . .

  tags = {
    Owner = element(var.owner_tag, count.index)
  }
  count   = length(var.instance_ips)
}
```

Now, if our `var.instance_ips` variable had `12` elements, then our `count` will create 12 instances. Terraform would select each element then wrap to the start of the list and select again. This way we'd end up with six instances tagged with `team1` and six instances tagged with `team2`.

Conditionals

The `count` meta-argument also allows us to explore Terraform's conditional logic. Terraform has a ternary operation conditional form.

NOTE Conditional logic was introduced in Terraform 0.8. It will not work in earlier releases.

A ternary operation looks like this:

Listing 3.74: A ternary operation

```
condition ? true : false
```

We specify a condition, followed by a `?`, and then the result to return if the condition is true or false, separated by `:`.

Let's see how we might use a conditional to set the `count` meta-argument as an alternative to the methods we've seen thus far.

Listing 3.75: Using ternary with count

```
variable "environment" {
  default = "development"
}

resource "aws_instance" "web" {
  ami           = lookup(var.ami, var.region)

. . .

  count         = var.environment == "production" ? 4 : 2
}
```

Here we've set a variable called `environment` with a default of `development`. In our resource we've configured our `count` attribute with a conditional. If the `var.environment` variable equals `production` then launch 4 instances, if it is the default of `development`, or any other value, then only launch 2 instances.

The condition can be any interpolation: a variable, a function, or even chaining another conditional. The true or false values can also return any interpolation or valid value. The true and false values must return the same type though.

The condition supports a bunch of operators. We've already seen equality, `==`, and Terraform supports the opposite operator `!=` for inequality. It also supports numeric comparisons like greater or less, `>` and `<`, and the related `>=` and `<=`. It also supports Boolean logic like: `&&`, `||` and unary `!`.

We don't have to use conditionals with just `count` though. They work on any resource or module attribute, for example:

Listing 3.76: A conditional attribute

```
module "vpc_basic" {
. . .

  cidr = var.region != "us-east-1" ? "172.16.0.0/12" :
"172.18.0.0/12"

. . .
}
```

Here we're setting the value of the `cidr` attribute using a ternary conditional. If the `var.region` variable is not equal to `us-east-1` then use the CIDR of `172.16.0.0/12`. If it is equal then use `172.18.0.0/12`.

TIP You can read more about conditionals in their documentation.

Locals

Terraform also has the concept of local value configuration. Local values assign a name to an expression, essentially allowing you to create repeatable function-like values.

NOTE Local values have been available since Terraform version 0.10.3.

We define local values in `locals` blocks.

Listing 3.77: A local definition

```
locals {
  instance_ip_count = length(var.instance_ips)
}
```

Here we've created a local value from the application of the `length` function to our `var.instance_ips` variable. This assigns to the resulting count of IPs in that variable to a local value of `instance_ip_count`. We can then use this local value in our resources without needing to repeat the function, for example:

Listing 3.78: Using a local in a resource

```
resource "aws_instance" "web" {
. . .

  tags = {
    Owner = element(var.owner_tag, count.index)
  }
  count   = instance_ip_count
}
```

Local expressions can refer to or use previously defined locals too but can't be self-referential. For example, you can't use a local within the expression that defines that local.

TIP A local is only available in the context of the module it is defined in. It will not work cross-module.

You can specify one or many `locals` blocks in a module. We'd recommend grouping them together for maintainability. If you use more than one `locals` block in a module then the names of the locals defined must be unique across the module.

Now let's look at provisioning some application configuration on our EC2 instances.

Provisioning our stack

Provisioning is the process of adding configuration, packages, applications, and services to the infrastructure we're creating. It usually involves making more granular changes to our infrastructure than we do with Terraform—for example, installing Apache on an EC2 instance. For complex provisioning we're likely to hand off the task to a dedicated tool like Puppet, Chef, or Ansible. For our stack, however, we're going to do some simple provisioning using EC2 user data. With user data, you can specify some commands or actions that should be run when the EC2 instance is launched.

TIP We'll learn more about provisioning and integration with configuration management tools in Chapter 4.

To make use of user data in Terraform we add the `user_data` attribute to our `aws_instance.web` resources in the `web.tf` file.

Listing 3.79: Adding user data to our instances

```
resource "aws_instance" "web" {

. . .
  user_data = file("files/web_bootstrap.sh")
. . .

  count     = length(var.instance_ips)
}
```

We can see that the value of our `user_data` attribute is:

`file("files/web_bootstrap.sh")`

This uses a new function, `file`, to load the contents of a file as the value of an attribute. In this case we're loading a shell script called `web_bootstrap.sh` from a directory called `files`. The location of the `files` directory is relative to the current directory.

Let's create that directory and file now.

Listing 3.80: Creating the files directory

```
$ pwd
~/terraform/web
$ mkdir files
$ cd files
$ touch web_bootstrap.sh
```

Let's add some commands to the `web_bootstrap.sh` script.

Listing 3.81: The web_bootstrap.sh script

```
#!/bin/bash
sudo apt-get update
sudo apt-get install -y nginx
sudo service nginx start
```

Now when our instances launch Nginx will automatically be installed and started. We'll see the results of this when we apply our configuration.

TIP It might take some time after the instance is launched to complete the installation process. Be patient! You can SSH into the instances to check the progress if required.

We're going to focus on more complex provisioning in Chapter 4. For now let's finish building our stack.

Finishing up our stack

At the bottom of our configuration file there's some security group configuration, providing security groups for some of the resources in our `web.tf` file. We're not going to show you this here because security group configuration is long and complex, but you can see it in the book's source code.

Finally, let's add some outputs to our stack. We'll create a new file in the `~/terraform/web` directory called `outputs.tf` and populate it.

Listing 3.82: The web outputs.tf file

```
output "elb_address" {
  value = aws_elb.web.dns_name
}

output "addresses" {
  value = aws_instance.web.[*].public_ip
}

output "public_subnet_id" {
  value = module.vpc_basic.public_subnet_id
}
```

We've specified three outputs. These outputs will be displayed at the end of our `terraform apply` run. We've specified the DNS name of our Elastic Load Balancer resource and a list of the public IP addresses of our EC2 instances. We've used the splat syntax of `[*]` to return the `public_ip` values of all of the EC2 instances we're going to create.

We've also specified an output that returns one of the outputs of the `vpc` module: `public_subnet_id`. Outputs allow us to bubble up attributes from all of our configurations, including modules.

Now let's tidy up a few loose ends by better managing our configuration.

Committing our configuration

Now is a good time to commit our configuration to Git. This will allow us to go back to a known good state if we need to, and to potentially share our configuration with others.

NOTE We're going to assume you know the basics of how Git works, and that you'll be regularly committing. This is just a reminder that it's a good idea to store your configuration in version control.

Listing 3.83: Committing our configuration

```
$ pwd
terraform/web
$ git add .
$ git commit -a "First draft of our web stack"
```

This will commit our current Terraform configuration to our Git repository. We could then push our configuration upstream to a shared repository for others to use.

Validating and formatting

Don't forget the `terraform validate` and `terraform fmt` commands we introduced in Chapter 2. The `validate` command checks the syntax, validates your Terraform configuration files, and returns any errors. The `fmt` command neatly formats your configuration files. These are both very useful, especially as your configurations get more complex.

Now let's see what happens when we plan the stack.

Initializing Terraform

Before we go any further we need to initialize this Terraform configuration and download our provider. We do this using the `terraform init` command.

Listing 3.84: Initialiazing the web configuration

```
$ terraform init
```

This will get our `aws` provider and update our local configuration.

Planning our stack

Now that our stack's configuration and initialization is complete we can build it. But before we do, it's always a good idea to run `terraform plan` to ensure the configuration is going to do what we expect.

Listing 3.85: Planning our web configuration

```
$ terraform plan
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but
will not be persisted to local or remote state storage.

. . .

+ aws_elb.web
    availability_zones.#:                       "<computed>"
    connection_draining:                        "false"
    connection_draining_timeout:                "300"
    cross_zone_load_balancing:                  "true"
    dns_name:                                   "<computed>"
    health_check.#:                             "<computed>"

. . .

+ module.vpc_basic.aws_internet_gateway.tfb
    tags.%:     "1"
    tags.Name: "web-igw"
    vpc_id:     "vpc-3c35d65a"

. . .

Plan: 9 to add, 0 to change, 0 to destroy.
```

We've included a sample of the `terraform plan` output. It shows us each resource that will be created, the values of the attributes that we know about now, and which of those will be computed when we apply the configuration. We can see that nine total resources will be created.

Now we're comfortable Terraform is going to do the right thing!

Applying our stack

Let's apply our execution plan and build our stack. To do this we run the `terraform apply` command.

NOTE In this command and future `terraform apply` commands we're going to skip the interactive prompt and assume you've typed `yes` to save some space in the output.

Listing 3.86: Applying our web stack

```
$ terraform apply
module.vpc_basic.aws_vpc.tfb: Creating...
  cidr_block:                "" => "10.0.0.0/16"
  default_network_acl_id:    "" => "<computed>"
  default_route_tablle_id:    "" => "<computed>"

. . .

  subnets.#:                                  "" => "1"
  subnets.248256935:                          "" => "subnet-8f0afcb3
"
  zone_id:                                    "" => "<computed>"
aws_elb.web: Creation complete

Apply complete! Resources: 9 added, 0 changed, 0 destroyed.

. . .

State path: terraform.tfstate

Outputs:

addresses = [
    54.167.183.26,
    54.167.186.170
]
elb_address = web-elb-1083111107.us-east-1.elb.amazonaws.com
public_subnet_id = subnet-ae6bacf5
```

It might take a couple minutes to create all of our configuration and finish. We should see that nine resources have been created. We can also see our outputs are the last items returned. We see the IP addresses of both our EC2 instances and the DNS name of our Elastic Load Balancer. We also see the ID of the `public` subnet we created with the `vpc` module.

If we want to see these outputs again, rather than applying the configuration again, we can run the `terraform output` command.

Listing 3.87: Showing the outputs only

```
$ terraform output
addresses = [
    54.167.183.26,
    54.167.186.170
]
elb_address = web-elb-1083111107.us-east-1.elb.amazonaws.com
public_subnet_id = subnet-ae6bacf5
```

TIP Remember if you want to see the full list of all our resources and their attributes you can run the `terraform show` command.

We can also make use of this data in other tools by outputting it in a machine-readable JSON format. To do this we can use the `terraform output` command with the `-json` flag.

Listing 3.88: Outputs as JSON

```
$ terraform output -json
{
    "addresses": {
        "sensitive": false,
        "type": "list",
        "value": [
            "54.167.183.26",
            "54.167.186.170"
        ]
    },
    "elb_address": {
        "sensitive": false,
        "type": "string",
        "value": "web-elb-1083111107.us-east-1.elb.amazonaws.com
"
    }
    "public_subnet_id": {
        "sensitive": false,
        "type": "string",
        "value": "subnet-ae6bacf5"
    }
}
```

We can consume this data in another service. For example, we could pass it to a provisioning tool such as Chef, Puppet, or Ansible.

Graphing our stack

Lastly, let's look at our stack's graph to see how the resources are interrelated. To output the graph we use the `terraform graph` command, pipe the result to a `.dot` file, and then convert it to an SVG file.

Listing 3.89: Graphing the web stack

```
$ terraform graph > web.dot
$ dot web.dot -Tsvg -o web.svg
```

We can then display the `web.svg` file.

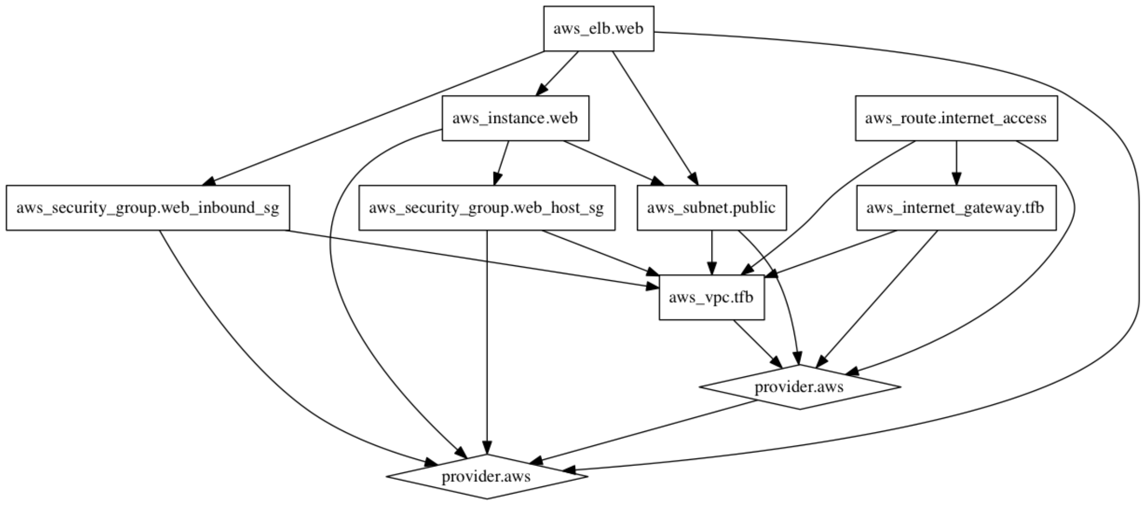


Figure 3.3: The graph of our web application stack

We can see two instances of the `aws` provider, one for our root configuration and the other for the `vpc` module. You can also see the relationships between the various resources we've just created.

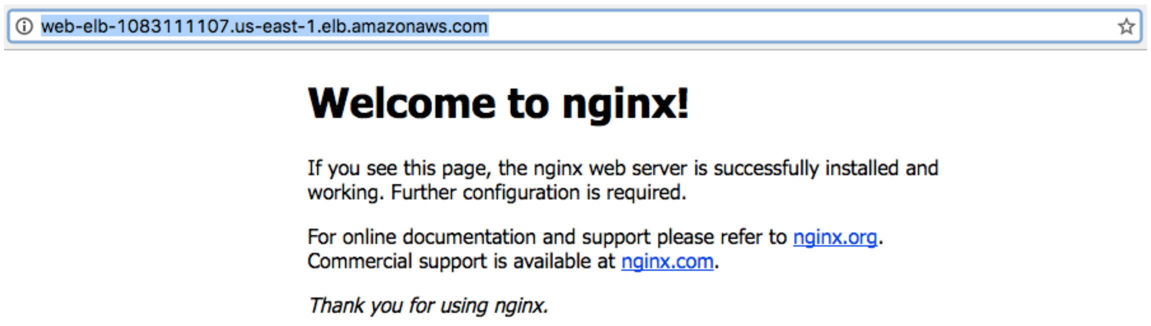
Seeing the results

Finally, we can actually see the results of our Terraform plan being executed by viewing the URL of the Elastic Load Balancer we just created. We can take the DNS name of the `aws_elb.web` resource from the outputs, in our case:

`web-elb-1083111107.us-east-1.elb.amazonaws.com`

We can browse to that URL and, if everything works, see the default Nginx index page.

TIP Remember, it might take a few minutes to complete the post-launch installation using our `user_data` script.



web-elb-1083111107.us-east-1.elb.amazonaws.com

Welcome to nginx!

If you see this page, the nginx web server is successfully installed and working. Further configuration is required.

For online documentation and support please refer to nginx.org.
Commercial support is available at nginx.com.

Thank you for using nginx.

Figure 3.4: Our stack in action

Voilà—we've created a simple, easily repeatable infrastructure stack!

TIP In addition to building a stack from your configuration, you can do the reverse and import existing infrastructure. You can read more about the import process in the Terraform documentation.

Destroying the web stack resources

If you're done with your web stack you can then destroy it (and stop spending any money on AWS resources) with the `terraform destroy` command.

Listing 3.90: Destroy our web stack

```
$ 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

module.vpc_basic.aws_vpc.tfb: Refreshing state... (ID: vpc-
a22c10c5)

. . .

module.vpc_basic.aws_vpc.tfb: Destruction complete

Destroy complete! Resources: 9 destroyed.
```

Now our web stack has been destroyed.

Summary

In this chapter we've put our burgeoning Terraform knowledge into action to build a simple web application with a load balancer. We've been introduced to the concept of parameterizing our configuration, allowing us to be more flexible in how we build our configuration. We've explored the types of variables available to us and how to use them.

We were introduced to modules, Terraform's approach to reusable infrastructure. We've learned how to build and use modules in our configuration. We've also learned about some of Terraform's meta-arguments. Finally, we learned about outputs and how to make use of them.

In the next chapter we'll learn more about how to provision software with Terraform, including how to connect it to existing provisioners like Puppet and Chef.

List of Figures

Listings

Index

Thanks! I hope you enjoyed the book.

ISBN 978-0-9888202-5-8 US$14.99
9 780988 820258 51499>