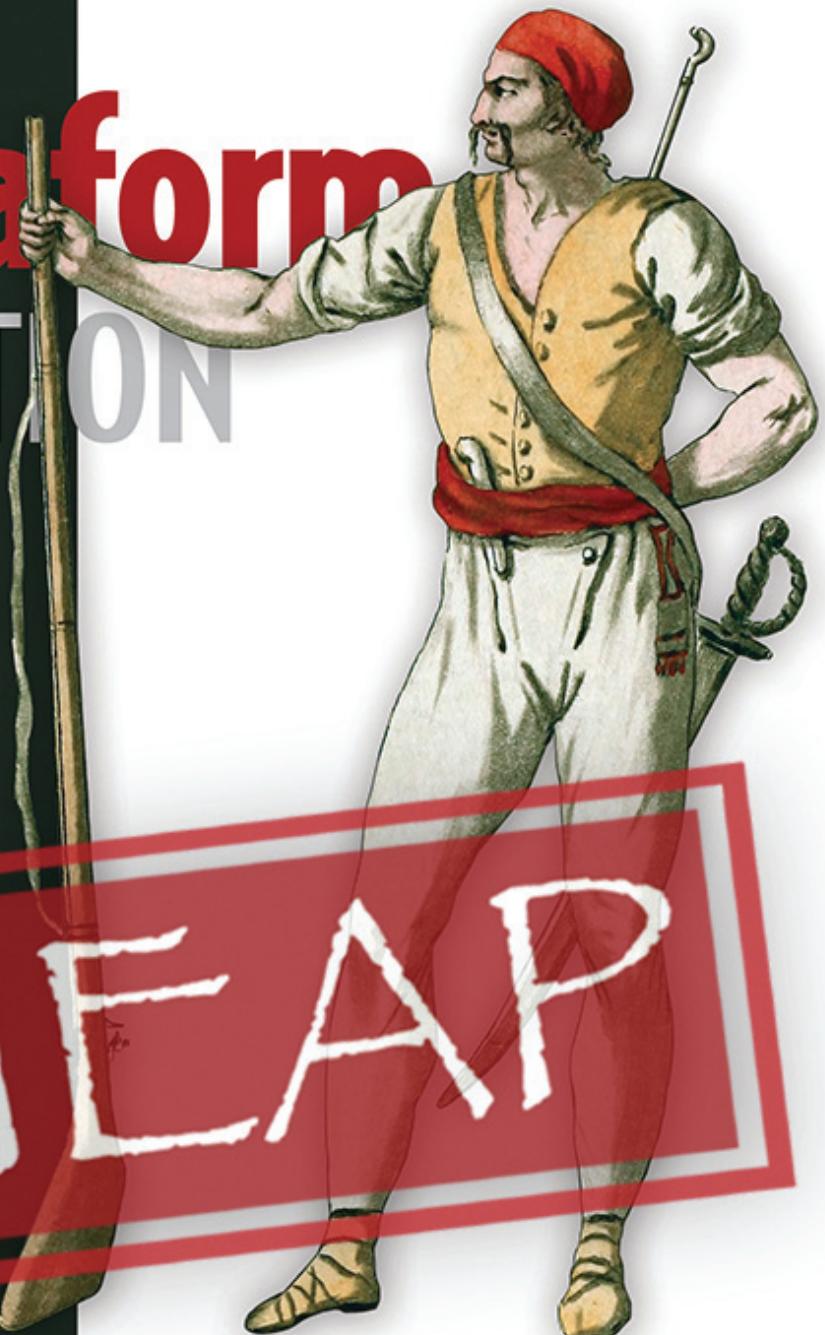


# Terraform IN ACTION

Scott Winkler



MANNING



**MEAP Edition  
Manning Early Access Program  
Terraform in Action  
Version 9**

Copyright 2020 Manning Publications

For more information on this and other Manning titles go to  
[manning.com](https://manning.com)

# welcome

---

Dear Reader,

Thank you for purchasing this *Terraform in Action*. Personally, I love books and I think they are the best way to broach a new subject. Now that I've written one of my own, I can understand why this is the case: an incredible amount of time and effort from many different people went into handcrafting this experience for you. If you want to jumpstart your skills with Terraform and skip all the boring stuff, then this is the book for you.

When I started learning Terraform there were not many resources available. In many ways, it was the Wild West of Terraform. My path to learning was long and tortuous, and so by writing this book, I hope to save you from going what I went through. This is a culmination of all my learnings and musings about Terraform over the years. I poured a lot of myself into this book and didn't hold anything back. If you can finish reading this then I promise you will know all that I know – you too can become a Terraform guru. My teaching style doesn't involve memorization or rote copy-paste, because I don't believe in that. Instead, I will teach you how to think in Terraform and how to solve your own problems. The world is a big place and the potential use cases of Terraform are far too many for this to merely be a cookbook.

Whether you are a beginner approaching Terraform for the first time, or already a seasoned expert, I believe there is something for you in this book. This is my gift to you, the reader. This is the book that I wish I had available to me when I was a beginner.

If you have any questions, or just want to share your thoughts, please send me an email to [scottjwinkler@gmail.com](mailto:scottjwinkler@gmail.com) or post any feedback you have in [liveBook discussion](#).

I look forward to hearing from you!

Thanks,  
—Scott

# *brief contents*

---

## **PART 1: LEARNING THE ROPES**

- 1 Getting Started with Terraform*
- 2 Lifecycle of a Terraform Resource*
- 3 Functional Programming*
- 4 Deploying a Multi-Tiered Web Application in AWS*

## **PART 2: TERRAFORM IN THE WILD**

- 5 Serverless Made Easy*
- 6 Terraform with Friends*
- 7 CI/CD Pipelines as Code*
- 8 A Multi-Cloud MMORPG*

## **PART 3: BECOMING A TERRAFORM GURU**

- 9 Zero Downtime Deployments*
  - 10 Refactoring and Testing*
  - 11 Extending Terraform by Writing your own Provider*
  - 12 Terraform in Automation*
  - 13 Secrets Management*
- Appendix A: Creating Custom Resources with the Shell Provider*

# 1

## *Getting Started with Terraform*

### **This chapter covers:**

- Why Terraform is so great
- Syntax of the HashiCorp Configuration Language (HCL)
- Fundamental elements and building blocks of Terraform
- Setting up a Terraform workspace
- Configuring and deploying an Ubuntu virtual machine on AWS

"*What does Terraform do?*", is a question I get asked frequently, but it's not one that I ever find easy to answer. At first, it seems like a no-brainer that should be obvious, but you'd be surprised how difficult it is to describe what Terraform is to someone who's never used it. When I was first asked this question, I replied that "Terraform is an Infrastructure as Code provisioning tool", because I knew that's what I was supposed to say, but I also didn't necessarily understand what any of that meant. While the phrase adequately describes what Terraform *is*, it's also not helpful for understanding what Terraform actually *does*. HashiCorp, the company behind Terraform, describes the technology as: "a tool for building, changing, and versioning infrastructure safely and efficiently", but even that isn't exactly clear. For starters, what does "infrastructure" mean? And what kind of problems does Terraform help you solve?

The word *infrastructure* is an elusive sort of word that seems to have taken on a multitude of different meanings in recent years. Infrastructure, at least in the tech sector, used to refer to the literal hardware components that you'd see in private, on-premise data centers. These data centers had racks of physical hardware components for the express purpose of running application software and making it available to the world. Again, these were physical devices, so you could walk through the racks of hardware components and feel the heat radiating off these machines, or just admire the flashing lights and beeping sounds as you walk around.

Nowadays, it's a different story. The cloud has well and truly taken over, and most companies have either migrated to the cloud or are in the process of migrating to the cloud. Tech giants like

Google, Amazon, and Microsoft have completely dominated the cloud scene by offering attractive on-demand compute, storage and networking related services via an *Application Programming Interface* (API). The word “infrastructure” thus no longer refers exclusively to physical hardware in on-premise data centers, as it could also refer to any virtualized hardware or managed software accessible through an API. What used to be a concrete and easily explainable term has now become abstract and muddied. As we shall see later, even this relaxed definition of “infrastructure” may not be appropriate in describing the current state of affairs.

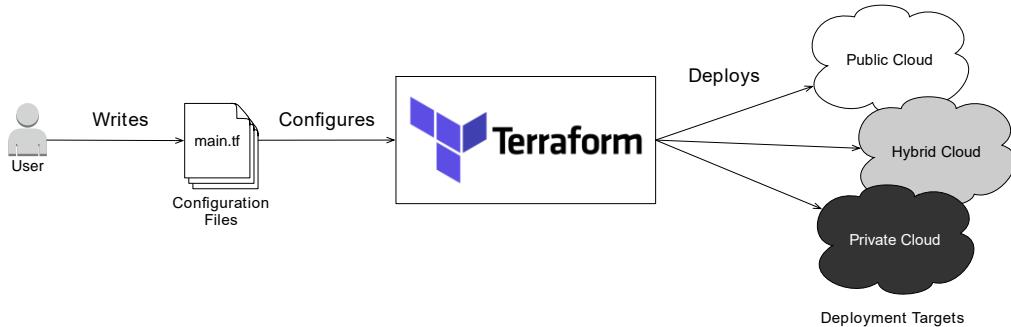
Terraform is an Infrastructure as Code technology created by HashiCorp for automating infrastructure provisioning to the cloud. Why the need for automation? Isn’t the cloud supposed to solve all our problems? Well, as it turns out, provisioning infrastructure to the cloud is a hugely tedious process, especially once you factor in the extreme scale that modern companies operate at. Companies used to have entire teams dedicated to the task of point-and-clicking their way through the console. Some of the more astute among those eventually discovered how to ~~do less work~~ increase productivity by writing custom scripts, which is really how the idea of Infrastructure as Code really started.

*Infrastructure as Code* (IaC) is the process of managing and provisioning data centers through machine-readable definition files, rather than manual point-and-click. It means writing code to define your infrastructure, in the same manner you would write code to define your application software. Expressing your infrastructure as code has many distinct advantages over manual provisioning, such as being able to check code into version control systems, perform regular audit checks, and deploy faster and more reliably.

Traditionally, IaC was accomplished using configuration management tools like Chef, Puppet, Ansible, and SaltStack. Configuration management tools like these provide a great deal of customization but tend to have a higher learning curve, and focus more on application delivery rather than infrastructure provisioning (which is where Terraform excels).

Terraform is an infrastructure provisioning tool, not a configuration management tool. Being a provisioning tool means that Terraform can deploy your entire infrastructure stack, not just new versions of your application (although it can do that too). By writing configuration files, Terraform can deploy just about anything. The main use case of Terraform is to deploy ephemeral, on-demand infrastructure in a predictable and repeatable fashion.

What can Terraform deploy to? Well, just about any cloud or combination of clouds, including: private, public, hybrid, and multi-cloud.



**Figure 1.1 Terraform can deploy infrastructure to any cloud or combination of clouds**

In this chapter we're going to start by going over the distinguishing features of Terraform. Terraform is just a tool after all, so why pick Terraform over another, when there are so many tools that can do the job? We'll also talk about the comparative advantages and disadvantages of Terraform in relation to other popular IaC technologies, and what makes Terraform the clear winner. Finally, we'll look at the quintessential "Hello World!" of Terraform by deploying a single server to AWS and improving it by incorporating some of the more dynamic features of Terraform.

## 1.1 What makes Terraform so great?

There's been a lot of hype about Terraform recently, but is any of it really justified? Terraform isn't the only Infrastructure as Code technology on the block. There are plenty of other tools that do the same thing. Moreover, Terraform was created and released by a relatively small tech company, and doesn't even have a 1.0 version out yet. How is it that Terraform, a technology in the highly lucrative software deployment market space, able to compete against competition from Amazon, Microsoft, and Google? The answer is, of course, that Terraform enjoys a unique set of competitive advantages. These competitive advantages stem from six key characteristics:

1. **Provisioning tool: Deploy infrastructure, not just applications**
2. **Easy to use:** For all of us non geniuses
3. **Free and Open Source:** Who doesn't like free?
4. **Declarative:** Say what you want, not how to do it
5. **Cloud agnostic:** Deploy to any cloud using the same tool.
6. **Expressive and extendable:** You aren't limited by the language

A comparison of Terraform to similar IaC tools is shown in table 1.1:

Name	Key Features					
	Provisioning tool	Easy to use	Free and Open Source	Declarative	Cloud Agnostic	Expressive and extendable

<b>Ansible<sup>1</sup></b>		X	X		X	X
<b>Chef<sup>2</sup></b>			X	X	X	X
<b>Puppet<sup>3</sup></b>			X	X	X	X
<b>SaltStack<sup>4</sup></b>		X	X	X	X	X
<b>Terraform<sup>5</sup></b>	X	X	X	X	X	X
<b>Pulumi<sup>6</sup></b>	X		X		X	X
<b>AWS CloudFormation<sup>7</sup></b>	X	X		X		
<b>GCP Resource Manager<sup>8</sup></b>	X	X		X		
<b>Azure Resource Manager<sup>9</sup></b>	X			X		

**Table 1.1 A comparison of popular IaC tools**

### 1.1.1 Provisioning Tool

Terraform is an infrastructure provisioning, not a configuration management, tool. The distinction is subtle but important. Configuration management (CM) tools, like Chef, Puppet, Ansible and SaltStack, were all designed to install and manage software on existing servers. They work by performing push or pull updates and restoring software to a desired state if drift has occurred. This works fine if you only care about application delivery, but it doesn't help at all with the initial infrastructure provisioning process. You'd still have to write custom scripts to deploy your infrastructure before configuration management tools could be of any use.

**NOTE** it's not a hard and fast rule that provisioning tools can only provision and configuration management tools can only configure. Many CM tools offer some degree of infrastructure provisioning, and vice versa.

The biggest difference between configuration management and provisioning tools is a matter of philosophy. CM tools favor mutable infrastructure, whereas Terraform and other provisioning tools, favor immutable infrastructure. What's the difference then between mutable and immutable infrastructure?

*Mutable infrastructure* is performing software updates in place, especially in the context of application delivery. It's having pet servers. If pet servers are sick, you don't kill them. You nurse them back to health by rolling out new software updates.

Immutable infrastructure, on the other hand, is about treating servers like cattle, not like pets. If a server gets into a bad or undesirable configuration state, the solution is to simply take it out back, shoot it in the head, and replace it with a new one.

<sup>1</sup>Ansible - <https://www.ansible.com>

<sup>2</sup>Chef - <https://www.chef.io>

<sup>3</sup>Puppet - <https://www.puppet.com/>

<sup>4</sup>SaltStack - <https://www.saltstack.com/>

<sup>5</sup>Terraform - <https://www.terraform.io/>

<sup>6</sup>Pulumi - <https://www.pulumi.com>

<sup>7</sup>CloudFormation - <https://aws.amazon.com/cloudformation/>

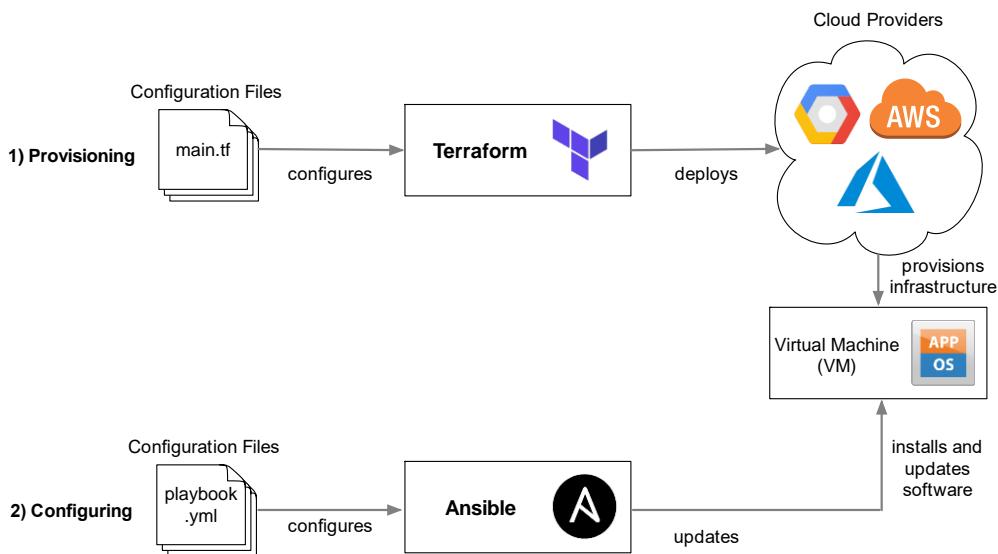
<sup>8</sup>GCP Resource Manager - <https://cloud.google.com/resource-manager/>

<sup>9</sup>Azure Resource Manager - <https://azure.microsoft.com/features/resource-manager/>

Another way to imagine the differences between mutable and immutable infrastructure is to think of an old toy that no longer does what you want. Mutable infrastructure would say that there is inherent goodness in the old toy and would recommend fixing it up. Immutable infrastructure, being significantly more callous and worldly, would suggest throwing out the old toy in favor of a new one that already has the features you want.

Why is immutability better than mutability? Because it's easier to reason about. You don't have to worry about things changing or leaving behind artifacts between deployments. Everything is commoditized, so there is no chance of having special snowflake servers. Additionally, this makes it easier to perform repeatable deployments and provision ephemeral environments that are all exactly uniform. Immutability is an ongoing trend in the software community. It's why we are moving away from virtual machines to containers, and from containers to serverless. The less you have to worry about the previous state of software, the better.

One conclusion that a lot of people make after first hearing about provisioning and configuration management tools is that they should be combined together. After all, why not use Terraform to deploy your infrastructure, and a configuration management tool to ensure it is in a desired state? See figure 1.2 for an example of how this would work.



**Figure 1.2 Combining Terraform and CM tools. Terraform deploy infrastructure initially and Ansible installs and updates software**

I am of the opinion that Terraform should not be combined with configuration management technologies, mainly because of their clashing philosophies. Terraform favors immutable infrastructure, whereas CM favors mutable. If you combine the two, invariably, you will wind up with something that straddles the paradigms of both, leading to something that is not much better than what we had before.

There are many ways to perform continuous delivery with Terraform that do not involve CM tools. For example, you could deploy new virtual machines with cloud-init scripts, or perform zero-downtime Blue/Green deployments<sup>10</sup>, or use a conventional container scheduler like Kubernetes/Nomad. The choice mainly has to do with what you are deploying, and the amount of time you are willing to spend to set it up. We'll talk more about these different strategies in chapters 7, 8, and 9.

### **1.1.2 Easy to Use**

The basics of Terraform are quick and easy to learn, even for non-programmers. By the end of chapter 4 you will already have the skills to call yourself an intermediate with the technology, which is kind of shocking when you think about it. Achieving mastery is another story, of course, but that's why you're here, and that's why this book is as long as it is.

The main reason why Terraform is so easy is because the code is written in a domain specific configuration language called *HashiCorp Configuration Language* (or just HCL for short). It's a language invented by HashiCorp as a substitute for more verbose configuration languages like JSON and XML. HCL attempts to strike a balance between human and machine readability and was influenced by earlier field attempts such as libuci and Nginx configuration. HCL is fully compatible with JSON, which means that HCL can be converted 1:1 to JSON, and vice-versa. This makes it easy to interoperate with systems outside of Terraform, or even generate configuration code on the fly.

### **1.1.3 Free and Open-Source Software**

The engine that powers Terraform is called *Terraform core*, a free and open source software offered under the Mozilla Public License v2.0. This license stipulates that anyone is allowed to use, distribute, or modify the software for both private and commercial purposes. Being free is great because you never have to worry about incurring additional costs when using Terraform. In addition, you gain full transparency to the product and are able to make open source contributions if you so desire. HashiCorp has a dedicated team working on the ongoing development of Terraform, and they are very good at fixing bugs and introducing new features in a timely manner.

There's also no premium version of Terraform that you're missing out by using the open source version. HashiCorp has repeatedly stated that they do not have any intentions to offer a premium version of Terraform, now or ever. Instead they plan to make money by offering specialized tooling that makes it easier to run Terraform in high-scale, multi-tenant environments. To this end, they have developed and released *Terraform Cloud* and *Terraform Enterprise*. Both products utilize the same open source Terraform core you can download for free, so it's easy to imagine developing your own platform if deploying Terraform workspaces. In chapter 11 we'll explore this topic, as well as discuss everything else you need to be aware of when running Terraform in automation.

### **1.1.4 Declarative Programming**

Declarative programming is when you expresses the logic of a computation (the what) without describing the control flow (the how). Instead of writing step-by-step instructions, you simply describe what you want, and it will get done. Examples of declarative programming languages include

<sup>10</sup> Blue/Green deployment is a technique that reduces downtime and risk by running two identical production environments called Blue and Green. At any time, only one of the environments is live, with the live environment serving all production traffic.

database query languages (SQL), functional programming languages (Haskell, Clojure), configuration languages (XML, JSON), and most IaC tools (Ansible, Chef, Puppet, Terraform).

Declarative programming is in contrast to imperative programming, which focuses on the how. With imperative programming, you use conditional branching, loops, and expressions to control system flow, save state, and execute commands. The hardware implementation of nearly all modern computers is imperative in nature, so it follows that most conventional programming languages are imperative as well (Python, Java, C, etc.).

The difference between declarative and imperative programming is the difference between getting in a car and driving yourself from point A to point B, vs. calling an Uber and having your chauffeur drive you. Assuming you are unfamiliar with the area, it is more than likely you will take an inefficient route, compared to your chauffeur.

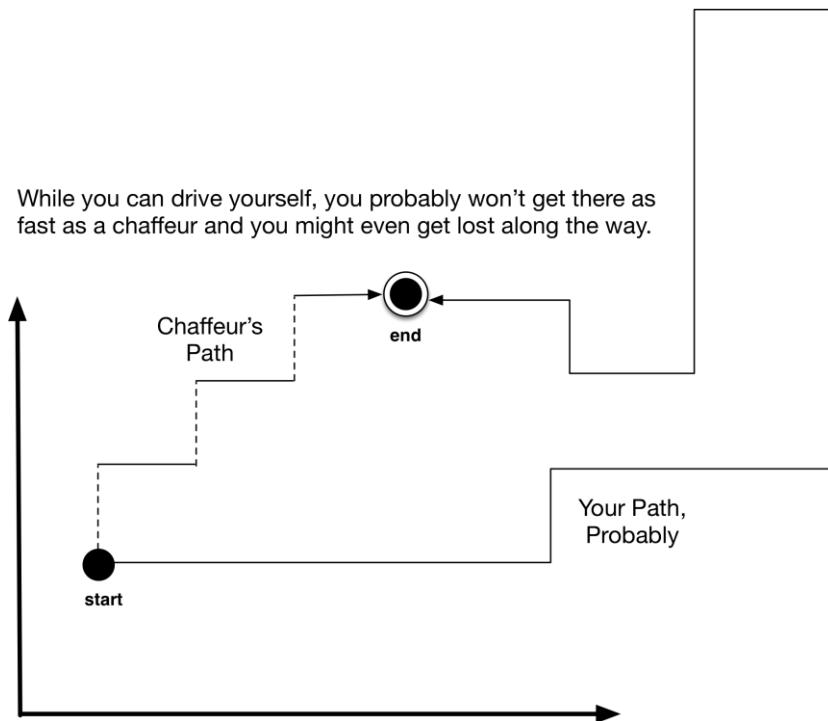


Figure 1.3 Driving yourself vs being chauffeured

**NOTE** Declarative programming cares about the destination, not the journey. Imperative programming cares about the journey, not the destination

### 1.1.5 Cloud Agnostic

Cloud agnostic means being able to seamlessly run on any cloud platform (private or public). Terraform is cloud agnostic because it allows you to deploy infrastructure to whichever cloud, or

combination of clouds using the same configuration language and deployment workflows. Being cloud agnostic is important because it means you aren't locked-in to a particular cloud vendor, and you don't have to learn a whole new technology if you do switch clouds.

The way Terraform integrates with all the different clouds is through *providers*. Providers are plugins for Terraform that are designed to interface with external APIs. Each cloud vendor maintains their own Terraform provider, enabling Terraform to interface with and manage resources in that cloud. Providers are written in golang and handle all of the procedural logic for authenticating, making API requests, handling timeouts and errors, and managing the lifecycle of resources. There are hundreds of providers available in the Terraform provider registry, together allowing Terraform to manage thousands of different kinds of resources. You can even write your own Terraform provider, which is something we will discuss in chapter 10.

Providers can be mixed and matched however you like. If you want to deploy to the hybrid or multi-cloud, it's almost as easy as deploying to the single cloud. There are architecture and design patterns to be aware of, but we will cover these in chapter 8.

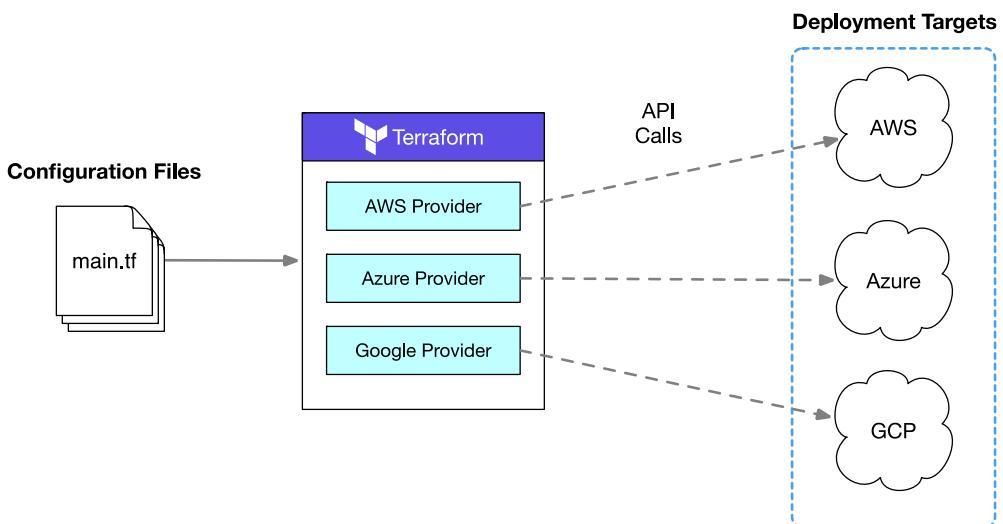


Figure 1.4 Deploying to multiple clouds at the same time with Terraform

### 1.1.6 Richly Expressive and Highly Extensible

Terraform is richly expressive and highly extensible compared to other declarative IaC tools. With conditionals, for expressions, directives, template files, dynamic blocks, variables, and many built-in functions, it's easy to write code to deploy exactly what you want. A tech comparison between Terraform and AWS CloudFormation is shown in table 1.2

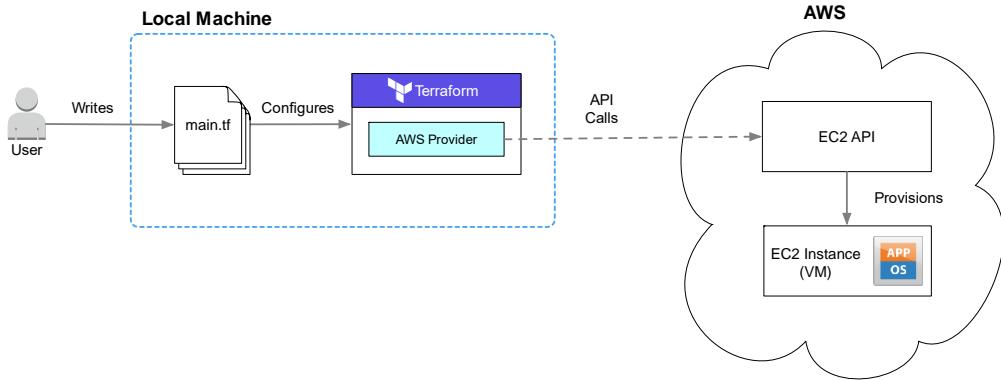
Name	Language Features				Other Features		
	Intrinsic Functions	Conditional Statements	For Loops	Types	Pluggable	Modular	Wait Conditions
Terraform	87	Yes	Yes	String, Number, List, Map, Boolean, Objects, Complex Types	Yes	Yes	No
AWS CloudFormation	11	Yes	No	String, Number, List	Limited	Yes	Yes

**Table 1.2 Tech comparison between the IaC tools Terraform and AWS CloudFormation**

You can also create reusable code by packaging them in modules, which can be shared with others through the public module registry (something we will discuss in chapter 6). Finally, as mentioned in the previous section, you can also extend Terraform by writing your own provider.

## 1.2 Hello Terraform!

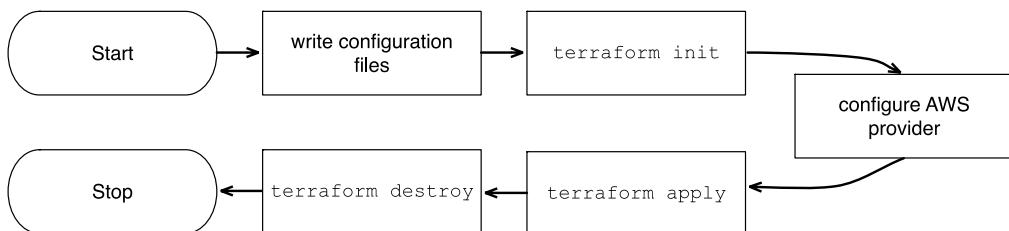
In this section we will take a look at a classical use case for Terraform - deploying a virtual machine (EC2 instance) onto AWS. We'll use the AWS provider for Terraform to make API calls on our behalf and deploy an EC2 instance. When we're done, we'll have Terraform take down the instance, so we don't incur ongoing costs by keeping the server running. An architecture diagram for what we are about to do is shown in figure 1.5.



**Figure 1.5 Using Terraform to deploy an EC2 instance to AWS**

As a prerequisite for this scenario, I except that you have Terraform 0.13.X installed<sup>14</sup>, and that you have access credentials for AWS. The steps we'll take for deploying the project are:

1. Writing Terraform configuration files
  2. Configuring the AWS provider
  3. Initializing Terraform with `terraform init`
  4. Deploying the EC2 instance with `terraform apply`
  5. Cleaning-up with `terraform destroy`
- This flow can be visualized by figure 1.6.



**Figure 1.6 sequence diagram of “Hello Terraform!” deployment**

### 1.2.1 Writing Terraform Configuration

Terraform reads from configuration files to deploy infrastructure. To tell Terraform we want it to deploy an EC2 instance, we need to declare an EC2 instance as code. Let's do that now. Start by creating a new file named `main.tf`, with the contents from Listing 1.1. The ".tf" extension signifies

<sup>14</sup>Install Terraform – <https://learn.hashicorp.com/terraform/getting-started/install.html>

that it's a Terraform configuration file. When Terraform runs it will read all files in the current working directory with a ".tf" extension and concatenate them together.

### **Listing 1.1 Contents of main.tf**

```
resource "aws_instance" "helloworld" { #A
    ami           = "ami-09dd2e08d601bff67" #B
    instance_type = "t2.micro" #B
    tags = { #B
        Name = "HelloWorld" #B
    } #B
}
```

#A Declaring an aws\_instance resource with name "helloworld"  
#B Attributes for the EC2 instance

**NOTE** this AMI is only valid for the us-west-2 region

This code in Listing 1.1 declares that we want Terraform to provision a t2.micro AWS EC2 instance with an Ubuntu AMI (Amazon Machine Image), and a name tag. Compare this to the equivalent CloudFormation code which is shown in Snippet 1.1, and you can see how much clearer and more concise it is.

### **Snippet 1.1 Equivalent AWS CloudFormation**

```
{
  "Resources": {
    "Example": {
      "Type": "AWS::EC2::Instance",
      "Properties": {
        "ImageId": "ami-09dd2e08d601bff67",
        "InstanceType": "t2.micro",
        "Tags": [
          {
            "Key": "Name",
            "Value": "HelloWorld"
          }
        ]
      }
    }
  }
}
```

The EC2 instance code block is a kind of Terraform `resource`. Resources are the most prevalent and important element in all of Terraform. They are how Terraform provisions infrastructure, such as virtual machines, load balancers, NAT gateways, databases, and so forth. Resources are declared as HCL objects, with type `resource` and exactly two labels. The first label specifies the type of resource you want to create, and the second is the resource name. Name has no special significance and is only used to reference the resource within a given module scope (we will talk about modules and module scope in chapter 4). Together, the type and name make up the `resource` identifier, which is unique for each resource. Figure 1.7 shows the syntax of a resource block.

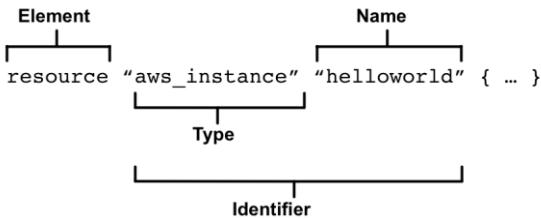


Figure 1.7 Syntax of a resource block

Each resource has inputs and outputs. Inputs are called *arguments* and outputs are called *attributes*. Arguments are passed through the resource and available as resource attributes, but there are also other attributes which you don't set and which are only available after the resource has been created. These other attributes are called *computed attributes* and are calculated information, typically metadata about the resource itself. Figure 1.8 shows some of the arguments, attributes and computed attributes of an `aws_instance` resource.

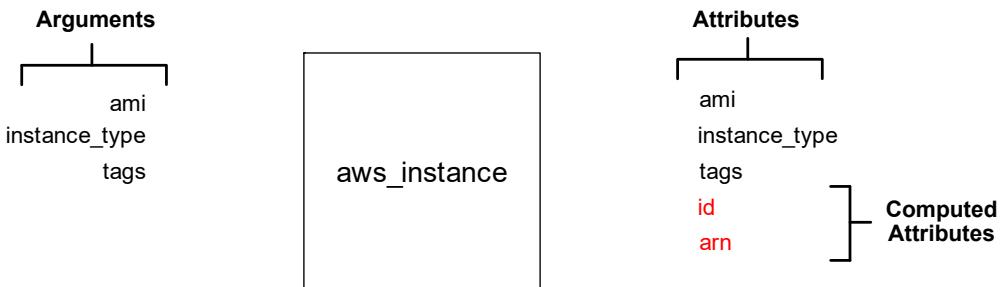


Figure 1.8 Inputs and outputs for an `aws_instance` resource

### 1.2.2 Configuring the AWS Provider

Next, we need to configure the AWS provider. The AWS provider is responsible for understanding API interactions, making authenticated requests, and exposing resources to Terraform – including the `aws_instance` resource. Let's explicitly declare our dependency on the AWS provider by adding a provider block to the configuration file. Update your code in `main.tf` to look like Listing 1.2

#### Listing 1.2 Contents of main.tf

```

provider "aws" { #A
  version = "2.65.0" #B
  region  = "us-west-2" #B
}

resource "aws_instance" "helloworld" {
  ami           = "ami-09dd2e08d601bff67"
  instance_type = "t2.micro"
}

```

```

tags = {
  Name = "HelloWorld"
}
}

```

#A Declaring the AWS provider

#B Configuring the provider by locking in a provider version and specifying a default region

## Authenticating to AWS

As mentioned previously, I expect you to already have default credentials for AWS. You can either store them in the `~/.aws/credentials` file, or as environment variables. There are also three other ways to authenticate, these being:

**Non-default profile credentials** – you can use profile credentials other than the default by setting an optional provider attribute known as `profile` (e.g. `profile = <your-profile-name>`)

**1) Static credentials** – you can hardcode AWS credentials in the Terraform AWS provider. This is a not a recommended, because of the risk of inadvertently exposing secret credentials through version control systems.

**2) Assumed role** – Terraform can use the credentials you give it to assume a role to another account and use that other role for all future actions.

If you need to modify the configuration of the AWS provider to fit your unique situation, I suggest consulting the AWS provider documentation<sup>12</sup> for more information on how to do one of the above listed options.

Unlike resources, providers only have one label: `name`. `Name` is the official name of the provider as published in the provider registry (e.g. “aws” for AWS, “google” for GCP and “azurerm” for Azure). The syntax for a provider block is shown in figure 1.9.

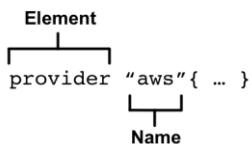


Figure 1.9 Syntax of a Provider Block

**NOTE** The provider registry is a global store for sharing versioned provider binaries. When Terraform initializes, it automatically looks up and downloads any required providers from the provider registry

Providers don't have outputs, only inputs. You configure a provider by passing in inputs, or *configuration arguments*, to the provider block. Configuration arguments will be things like service endpoint URL, region, provider version, and any credentials needed to authenticate against the API. Providers use the configuration arguments to configure all the resources underneath them. This can be visualized by figure 1.10.

<sup>12</sup> <https://www.terraform.io/docs/providers/aws/index.html>

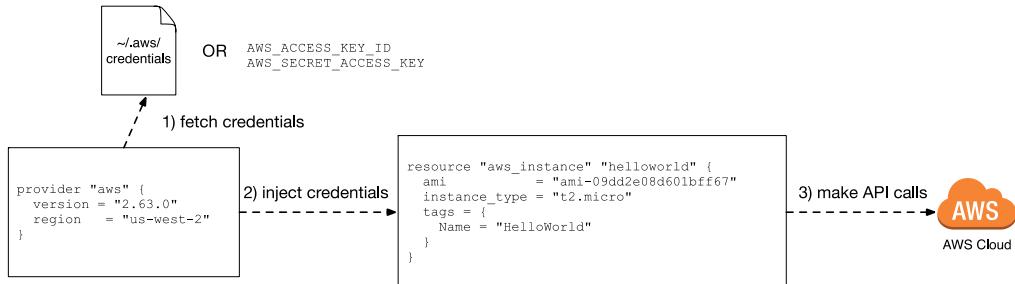


Figure 1.10 How the configured provider injects credentials into `aws_instance` when making API calls

Usually you don't want to pass secrets into the provider as plain text, especially when this code will later be checked into version control, so many providers have the ability to read environment variables or shared credential files to retrieve this secret information dynamically. If you are interested more in secrets management, I recommend reading chapter 12, which is where we cover this topic in great detail.

### 1.2.3 Initializing Terraform

Before we have Terraform deploy our EC2 instance, we first have to initialize the workspace. Even though we have declared the AWS provider, Terraform still needs to download and install the binary from the provider registry. Initialization is required for all new workspaces, as well as any new Terraform project you download from source control and are running for the first time.

You can initialize Terraform by running the command: `terraform init`. When you do this, you will see the following output:

**NOTE** you will need to have Terraform installed on your machine for this to work, if you do not have it already

```
$ terraform init

Initializing the backend...

Initializing provider plugins...
- Finding latest version of hashicorp/aws...
- Installing hashicorp/aws v2.65.0... #A
- Installed hashicorp/aws v2.65.0 (signed by HashiCorp)

The following providers do not have any version constraints in configuration,
so the latest version was installed.

To prevent automatic upgrades to new major versions that may contain breaking
changes, we recommend adding version constraints in a required_providers block
in your configuration, with the constraint strings suggested below.

* hashicorp/aws: version = "~> 2.65.0"

Terraform has been successfully initialized! #B

You may now begin working with Terraform. Try running "terraform plan" to see
```

any changes that are required for your infrastructure. All Terraform commands should now work.

If you ever set or change modules or backend configuration for Terraform, rerun this command to reinitialize your working directory. If you forget, other commands will detect it and remind you to do so if necessary.

#A Here Terraform is fetching the latest version of the AWS provider  
#B The only thing that we really care about

#### 1.2.4 Deploying the EC2 Instance

Now we're ready to deploy the EC2 instance using Terraform. Do this now by executing the `terraform apply` command.

**WARNING** performing this action may result in charges to your AWS account. This include potential charges for EC2 and CloudWatch Logs.

```
$ terraform apply
```

An execution plan has been generated and is shown below.  
Resource actions are indicated with the following symbols:  
+ create

Terraform will perform the following actions:

```
# aws_instance.helloworld will be created
+ resource "aws_instance" "helloworld" { #A
    + ami                      = "ami-09dd2e08d601bfff67" #B
    + arn                      = (known after apply)
    + associate_public_ip_address = (known after apply)
    + availability_zone          = (known after apply)
    + cpu_core_count             = (known after apply)
    + cpu_threads_per_core       = (known after apply)
    + get_password_data          = false
    + host_id                   = (known after apply)
    + id                        = (known after apply)
    + instance_state              = (known after apply)
    + instance_type                = "t2.micro" #C
    + ipv6_address_count         = (known after apply)
    + ipv6_addresses              = (known after apply)
    + key_name                   = (known after apply)
    + network_interface_id        = (known after apply)
    + outpost_arn                 = (known after apply)
    + password_data               = (known after apply)
    + placement_group              = (known after apply)
    + primary_network_interface_id = (known after apply)
    + private_dns                  = (known after apply)
    + private_ip                   = (known after apply)
    + public_dns                   = (known after apply)
    + public_ip                     = (known after apply)
    + security_groups              = (known after apply)
    + source_dest_check            = true
    + subnet_id                   = (known after apply)
    + tags                         = { #D
        + "Name" = "HelloWorld"
    }
}
```

```

        }
+ tenancy           = (known after apply)
+ volume_tags      = (known after apply)
+ vpc_security_group_ids = (known after apply)

+ ebs_block_device {
    + delete_on_termination = (known after apply)
    + device_name          = (known after apply)
    + encrypted            = (known after apply)
    + iops                 = (known after apply)
    + kms_key_id           = (known after apply)
    + snapshot_id          = (known after apply)
    + volume_id             = (known after apply)
    + volume_size           = (known after apply)
    + volume_type           = (known after apply)
}

+ ephemeral_block_device {
    + device_name   = (known after apply)
    + no_device     = (known after apply)
    + virtual_name  = (known after apply)
}

+ metadata_options {
    + http_endpoint      = (known after apply)
    + http_put_response_hop_limit = (known after apply)
    + http_tokens         = (known after apply)
}

+ network_interface {
    + delete_on_termination = (known after apply)
    + device_index          = (known after apply)
    + network_interface_id  = (known after apply)
}

+ root_block_device {
    + delete_on_termination = (known after apply)
    + device_name          = (known after apply)
    + encrypted            = (known after apply)
    + iops                 = (known after apply)
    + kms_key_id           = (known after apply)
    + volume_id             = (known after apply)
    + volume_size           = (known after apply)
    + volume_type           = (known after apply)
}
}

```

**Plan:** 1 to add, 0 to change, 0 to destroy. #D

**Do you want to perform these actions?**

Terraform will perform the actions described above.  
Only 'yes' will be accepted to approve.

Enter a value: #E

#A plus sign means create  
#B ami attribute  
#C instance\_type attribute  
#D tags attribute

```
#D Totals summary
#E Manual approval step
```

**TIP** If you received an error saying “No Valid Credentials Sources Found” then you need to make sure you have either default credentials set in `~/.aws/credentials`, or `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` set.

The output is called an *execution plan* and it outlines the set of actions that Terraform intends to perform to achieve your desired state. It’s a good idea to review the plan as a sanity check before proceeding. There shouldn’t be anything odd here, unless you made a typo, so when you are done reviewing the execution plan, approve it by entering “yes” at the command line.

After a minute or two (the approximate time it takes to provision an EC2 instance) the apply will complete successfully. An example output is shown below.

```
aws_instance.helloworld: Creating...
aws_instance.helloworld: Still creating... [10s elapsed]
aws_instance.helloworld: Still creating... [20s elapsed]
aws_instance.helloworld: Still creating... [30s elapsed]
aws_instance.helloworld: Creation complete after 34s [id=i-0c98c98679298dab8]

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

You can verify your resource was created by finding it in the AWS console for EC2, as shown in figure 1.11. Note that this instance will be in the us-west-2 region, because that’s what we set in the provider.

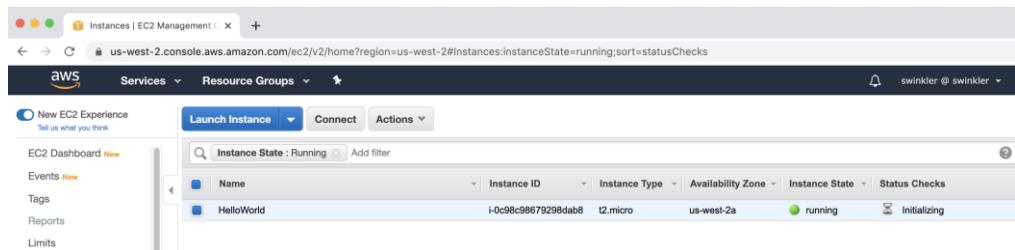


Figure 1.11 Screenshot of EC2 instance in AWS Console

All of the stateful information about the resource is stored in a file called `terraform.tfstate`. Don’t let the “`.tfstate`” prefix fool you, it’s really just a JSON file. The `terraform show` command can be used to print human readable output from the state file and makes it easy to list all the data of the resources Terraform manages. The show command and output result is shown below.

```
$ terraform show
# aws_instance.helloworld:
resource "aws_instance" "helloworld" {
    ami                  = "ami-09dd2e08d601bff67"
    arn                 = "arn:aws:ec2:us-west-2:215974853022:instance/i-
                           0c98c98679298dab8"
    associate_public_ip_address = true
```

```

availability_zone          = "us-west-2a"
cpu_core_count             = 1
cpu_threads_per_core       = 1
disable_api_termination    = false
ebs_optimized              = false
get_password_data          = false
hibernation                = false
id                         = "i-0c98c98679298dab8" #A
instance_state              = "running"
instance_type                = "t2.micro"
ipv6_address_count          = 0
ipv6_addresses              = []
monitoring                  = false
primary_network_interface_id = "eni-003e10654e38f1447"
private_dns                  = "ip-172-31-28-98.us-west-2.compute.internal"
private_ip                   = "172.31.28.98"
public_dns                   = "ec2-52-38-134-200.us-west-2.compute.amazonaws.com"
public_ip                     = "52.38.134.200"
security_groups              = [
    "default",
]
source_dest_check            = true
subnet_id                    = "subnet-0d78ac285558cff78"
tags                         = {
    "Name" = "HelloWorld"
}
tenancy                      = "default"
volume_tags                  = {}
vpc_security_group_ids       = [
    "sg-0d8222ef7623a02a5",
]
credit_specification {
    cpu_credits = "standard"
}

metadata_options {
    http_endpoint           = "enabled"
    http_put_response_hop_limit = 1
    http_tokens              = "optional"
}

root_block_device {
    delete_on_termination = true
    device_name           = "/dev/sda1"
    encrypted              = false
    iops                  = 100
    volume_id              = "vol-05366f9f647f56541"
    volume_size             = 8
    volume_type             = "gp2"
}
}

```

#A id is an important computed attribute

There's a lot more data here than we originally set in the resource block. The reason for this is because most of the arguments are either optional or computed after the resource has been created. You can customize `aws_instance` by passing in optional arguments. If you want to know what these

are, I recommend looking through the AWS provider documentation, as well as the AWS API documentation.

### 1.2.5 Destroying the EC2 Instance

Now it's time to say goodbye to the EC2 instance. You always want to take down any infrastructure you are no longer using, as it costs money to run stuff in the cloud. Terraform has a special command to destroy all resources, called `terraform destroy`. When you run this command, you will be prompted to manually confirm the destroy operation, as shown below.

```
$ terraform destroy
aws_instance.helloworld: Refreshing state... [id=i-0c98c98679298dab8]
```

An execution plan has been generated and is shown below.  
Resource actions are indicated with the following symbols:

- destroy

Terraform will perform the following actions:

```
# aws_instance.helloworld will be destroyed
- resource "aws_instance" "helloworld" { #A
    - ami                      = "ami-09dd2e08d601bff67" -> null
    - arn                      = "arn:aws:ec2:us-west-2:215974853022:instance/i-
      0c98c98679298dab8" -> null
    - associate_public_ip_address = true -> null
    - availability_zone        = "us-west-2a" -> null
    - cpu_core_count           = 1 -> null
    - cpu_threads_per_core     = 1 -> null
    - disable_api_termination   = false -> null
    - ebs_optimized            = false -> null
    - get_password_data        = false -> null
    - hibernation              = false -> null
    - id                       = "i-0c98c98679298dab8" -> null
    - instance_state            = "running" -> null
    - instance_type             = "t2.micro" -> null
    - ipv6_address_count       = 0 -> null
    - ipv6_addresses            = [] -> null
    - monitoring               = false -> null
    - primary_network_interface_id = "eni-003e10654e38f1447" -> null
    - private_dns               = "ip-172-31-28-98.us-west-2.compute.internal" -> null
    - private_ip                 = "172.31.28.98" -> null
    - public_dns                = "ec2-52-38-134-200.us-west-2.compute.amazonaws.com" ->
      null
    - public_ip                  = "52.38.134.200" -> null
    - security_groups           = [
        - "default",
    ] -> null
    - source_dest_check         = true -> null
    - subnet_id                 = "subnet-0d78ac285558cff78" -> null
    - tags                      = {
        - "Name" = "HelloWorld"
    } -> null
    - tenancy                   = "default" -> null
    - volume_tags               = {} -> null
    - vpc_security_group_ids    = [
        - "sg-0d8222ef7623a02a5",
    ] -> null
}
```

```

- credit_specification {
  - cpu_credits = "standard" -> null
}

- metadata_options {
  - http_endpoint           = "enabled" -> null
  - http_put_response_hop_limit = 1 -> null
  - http_tokens              = "optional" -> null
}

- root_block_device {
  - delete_on_termination = true -> null
  - device_name           = "/dev/sda1" -> null
  - encrypted              = false -> null
  - iops                   = 100 -> null
  - volume_id               = "vol-05366f9f647f56541" -> null
  - volume_size              = 8 -> null
  - volume_type              = "gp2" -> null
}
}

```

**Plan:** 0 to add, 0 to change, 1 to destroy. #B

**Do you really want to destroy all resources?**

Terraform will destroy all your managed infrastructure, as shown above.

There is no undo. Only 'yes' will be accepted to confirm.

**Enter a value:**

#A minus sign means destroy

#B Summary of all the actions Terraform intends to take

**WARNING** It is important not to manually edit or delete the `terraform.tfstate` file, or else Terraform will lose track of all managed resources

The destroy plan is just like the previous execution plan, except it is for the delete operations instead of create.

**NOTE** `terraform destroy` does exactly the same thing as if you were to delete all the configuration code and run a `terraform apply`.

Confirming that you wish to apply the destroy plan by typing in "yes" at the prompt, wait a few minutes for Terraform to do its thing, and you will be notified that Terraform has finished destroying all resources,. Your output will look like the following:

```

aws_instance.helloworld: Destroying... [id=i-0c98c98679298dab8]
aws_instance.helloworld: Still destroying... [id=i-0c98c98679298dab8, 10s elapsed]
aws_instance.helloworld: Still destroying... [id=i-0c98c98679298dab8, 20s elapsed]
aws_instance.helloworld: Still destroying... [id=i-0c98c98679298dab8, 30s elapsed]
aws_instance.helloworld: Destruction complete after 31s

```

**Destroy complete! Resources: 1 destroyed.**

You can verify that the resources have indeed been destroyed either by refreshing the AWS console, or running a `terraform show` and confirming that it returns nothing.

```
$ terraform show
```

## 1.3 Brave New Hello World!

I like the classic “Hello World!” example, and feel it is a good starter project, but I also don’t think it does justice to the technology as a whole. Terraform can do much more than simply provision resources from static configuration code. It’s able to provision resources dynamically based on the results of external queries and data lookups. Let us now consider *data sources*, which are elements that allow you to fetch data at runtime and perform computations as well.

In this section we will improve the classic “Hello World!” example by adding a data source to dynamically lookup the latest value of the Ubuntu AMI. We’ll pass the output value into `aws_instance` so we don’t have to statically set the AMI statically in the EC2 instance resource configuration (see figure 1.12).

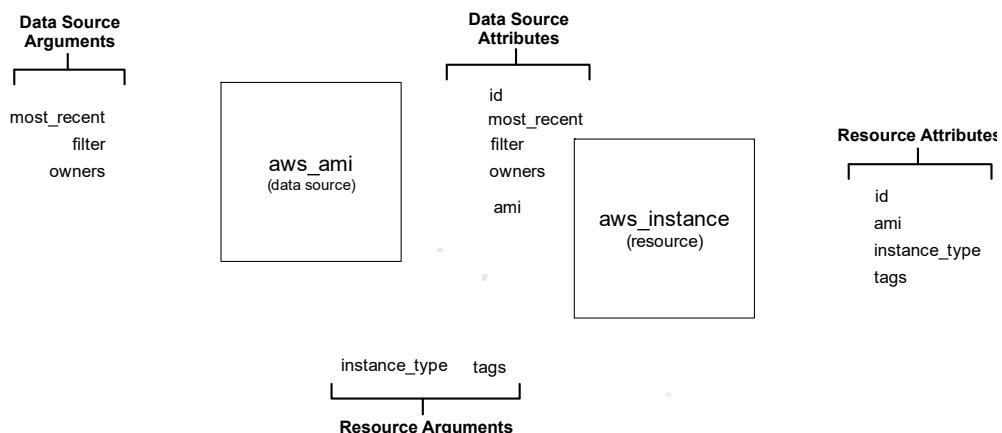


Figure 1.12 How the output of the `aws_ami` data source will be chained to the input of the `aws_instance` resource

Because we’ve already configured the AWS provider and initialized Terraform with `terraform init`, we can skip some of the steps we did previously. Here, we’ll be doing the following:

1. Modifying Terraform configuration to add the data source
2. Re-deploying with `terraform apply`
3. Cleaning-up with `terraform destroy`

This flow can also be visualized by figure 1.13.

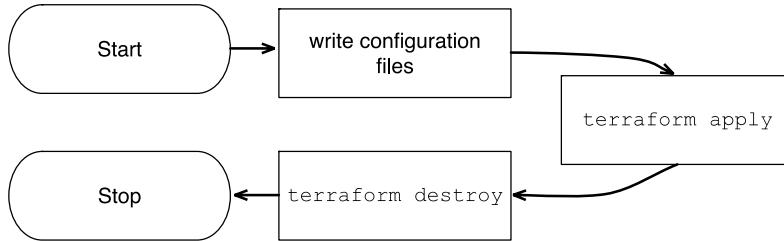


Figure 1.13 sequence diagram of deployment

### 1.3.1 Modifying Terraform Configuration

We need to add the code to read from the external data source, allowing us to query the most recent Ubuntu AMI published to AWS. Edit the `main.tf` to look like Listing 1.3.

#### Listing 1.3 Contents of main.tf

```

provider "aws" {
  version = "2.65.0"
  region = "us-west-2"
}

data "aws_ami" "ubuntu" { #A
  most_recent = true

  filter { #B
    name   = "name"
    values = ["ubuntu/images/hvm-ssd/ubuntu-focal-20.04-amd64-server-*"]
  }

  owners = ["099720109477"] #C
}

resource "aws_instance" "helloworld" {
  ami           = data.aws_ami.ubuntu.id #D
  instance_type = "t2.micro"
  tags = {
    Name = "HelloWorld"
  }
}

```

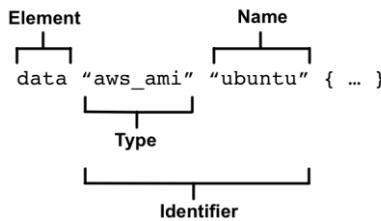
#A Declaring an aws\_ami data source with name "ubuntu"

#B Setting a filter to select all AMI's with name matching this regex expression

#C canonical Ubuntu AWS account id

#D chaining resources together

Like resources, data sources are declared by creating an HCL object with type “data” having exactly two labels. The first label specifies the type of data source, and the second is the name of the data source. Together the type and name are referred to as the data source’s “identifier” and must be unique within a module. Figure 1.14 illustrates the syntax of a data source.



**Figure 1.14 Syntax of a data source**

The contents of a data source code block are called *query constraint arguments* and behave exactly the same as arguments do for resources. The query constraint arguments are used to specify which resource(s) to fetch data from. Data sources are *unmanaged resources* that Terraform can read data from but doesn't directly control. This is in contrast to *managed resources*, which are resources that Terraform directly controls the entire lifecycle of.

### 1.3.2 Applying Changes

Let's go ahead and apply our changes by having Terraform deploy an EC2 instance with the Ubuntu data source output value for AMI. Do this now by running `terraform apply`. Your log output will be as follows.

```
$ terraform apply
data.aws_ami.ubuntu: Refreshing state... #A

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
+ create

Terraform will perform the following actions:

# aws_instance.helloworld will be created
+ resource "aws_instance" "helloworld" {
  + ami                               = "ami-09dd2e08d601bff67" #B
  + arn                             = (known after apply)
  + associate_public_ip_address     = (known after apply)
  + availability_zone                = (known after apply)
  + cpu_core_count                  = (known after apply)
  + cpu_threads_per_core           = (known after apply)
  + get_password_data              = false
  + host_id                         = (known after apply)
  + id                             = (known after apply)
  + instance_state                  = (known after apply)
  + instance_type                  = "t2.micro" #C

  // skip some logs
}

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

**Do you want to perform these actions?**  
Terraform will perform the actions described above.

```
Only 'yes' will be accepted to approve.
```

**Enter a value:**

```
#A read latest Ubuntu AMI
#B set from output of data source
```

Apply the changes by entering "yes" into the command line. After waiting a few minutes, your output will be:

```
aws_instance.helloworld: Creating...
aws_instance.helloworld: Still creating... [10s elapsed]
aws_instance.helloworld: Still creating... [20s elapsed]
aws_instance.helloworld: Still creating... [30s elapsed]
aws_instance.helloworld: Creation complete after 34s [id=i-07cf7ed8831c72324]
```

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

As before, you can verify the changes for yourself by either navigating through the AWS console, or invoking `terraform show`.

### 1.3.3 Destroying Infrastructure

Destroy the infrastructure that we created in the previous step by running `terraform destroy`. You'll receive another manual confirmation:

```
$ terraform destroy
...
  - root_block_device {
      - delete_on_termination = true -> null
      - device_name          = "/dev/sda1" -> null
      - encrypted            = false -> null
      - iops                 = 100 -> null
      - volume_id             = "vol-05366f9f647f56541" -> null
      - volume_size           = 8 -> null
      - volume_type           = "gp2" -> null
    }
}

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

**Do you really want to destroy all resources?**

Terraform will destroy all your managed infrastructure, as shown above.  
There is no undo. Only 'yes' will be accepted to confirm.

**Enter a value:**

After manually confirming and waiting a few more minutes, the EC2 instance is now gone.

```
aws_instance.example: Destroying... [id=i-07cf7ed8831c72324]
aws_instance.example: Still destroying... [id=i-07cf7ed8831c72324, 10s elapsed]
aws_instance.example: Still destroying... [id=i-07cf7ed8831c72324, 20s elapsed]
aws_instance.example: Still destroying... [id=i-07cf7ed8831c72324, 30s elapsed]
aws_instance.example: Destruction complete after 30s
```

```
Destroy complete! Resources: 1 destroyed.
```

## 1.4 Fireside Chat

In this introductory chapter, not only did we discuss what Terraform is, and how it compares to other IaC tools, we also performed two real-world deployments. The first is the de-facto “Hello World!” of Terraform, while the second is my personal favorite, because it throws in an additional data source to demonstrate the dynamic capabilities of Terraform.

In the next few chapters, we will go through the fundamentals of how Terraform works, as well as the major constructs and syntax elements of the Terraform HCL language. This builds up to chapter 4 when we deploy a complete load balanced web server and application onto AWS. Stick with it, and I promise it will be worth it!

## 1.5 Summary

- Terraform is a declarative IaC provisioning tool. It can deploy resources onto any public or private cloud.
- Terraform is: 1) a provisioning tool 2) easy to use 3) free and open source 4) declarative 5) cloud agnostic 6) expressive and extensible
- The major configuration elements of Terraform are resources, data sources and providers
- Code blocks can be chained together to perform dynamic deployments
- To deploy a Terraform project you first write configuration code, then configure providers and other input variables, `terraform init`, and finally `terraform apply`. Clean-up is done with `terraform destroy`.

# 2

## *Lifecycle of a Terraform Resource*

### **This chapter covers:**

- Generating and applying execution plans
- Analyzing when function hooks are triggered by Terraform
- Utilizing the Local provider to create and manage files
- Simulating, detecting, and correcting for configuration drift
- Understanding the basics of Terraform state management

When you do away with all its bells and whistles, Terraform is a surprisingly simple technology. Fundamentally, Terraform is a glorified state management tool that performs CRUD operations (create, read, update, delete) on managed resources. Oftentimes managed resources will be cloud-based resources, but they don't have to be. Anything that implements CRUD can be represented as a Terraform resource, so long as there is sufficient desire and motivation to do so.

In this chapter we will deep-dive into the internals of Terraform by walking through the lifecycle of a single resource. We could use any resource for this task, but to keep things simple, we'll use a resource that doesn't call any remote network APIs. These special sorts of resources are called *local-only resources* and only exist within the confines of Terraform, or the machine running Terraform. Local-only resources typically serve a marginal purpose, such as to glue together "real" infrastructure objects, but they also make a great teaching aid. Examples of local-only resources include private keys, self-signed TLS certificates, and random ids.

We will be using the `local_file` resource from the Local provider for Terraform to create, read, update, and delete a text file containing the first few passages of Sun Tzu's, "The Art of War".

## 2.1 Process Overview

`local_file` is a resource from the Local provider, and it allows us to create and manage text files with Terraform. We'll use it to create an `art_of_war.txt` file containing the first two stanzas of Sun Tzu's, "The Art of War". Our high-level architecture diagram is illustrated in figure 2.1.

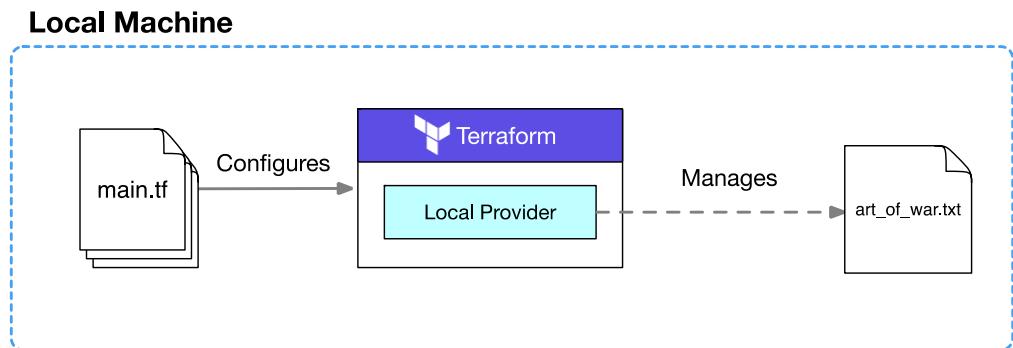


Figure 2.1 Inputs and outputs of the Sun Tzu scenario

**NOTE** Although a text file isn't normally considered "infrastructure", you can still deploy it in the same way you would an EC2 instance. Does that mean that it's "infrastructure"? Does the distinction even matter anymore? I'll leave it for you to decide.

First, we'll create the resource. Next, we'll simulate configuration drift and perform an update. Finally, we'll clean up with a `terraform destroy`. Our procedure is shown in figure 2.2.

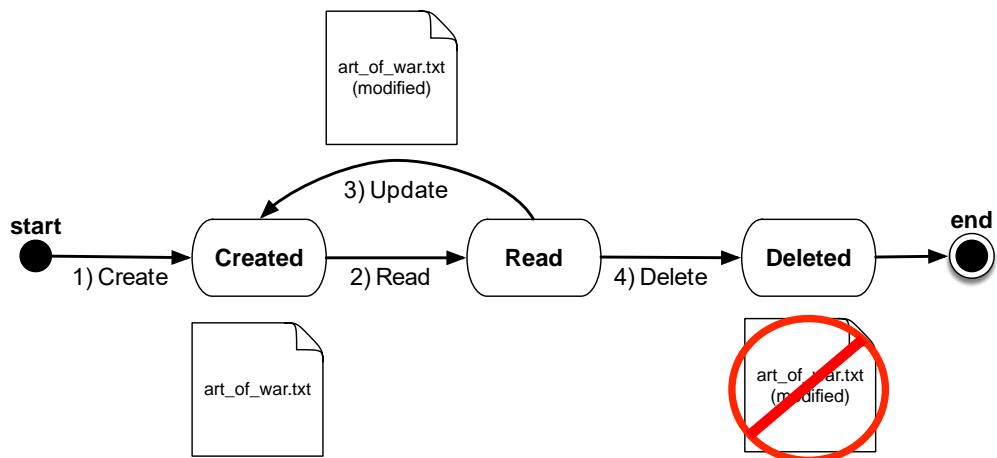


Figure 2.2 First, we create the resource [1], then we read and update it [2] and [3]. Finally, we delete the resource [4].

### 2.1.1 Lifecycle Function Hooks

All Terraform resources implement the resource schema interface. The resource schema mandates, among other things, that resources define CRUD functions hooks, one each for `Create()`, `Read()`, `Update()`, and `Delete()`. Terraform invokes these hooks when certain conditions are met. Generally speaking, `Create()` is called during resource creation, `Read()` during plan generation, `Update()` during resource updates, and `Delete()` during deletes. There's a bit more to it than that, but you get the idea.

Because it's a resource, `local_file` also implements the resource schema interface. That means that it defines function hooks for `Create()`, `Read()`, `Update()`, and `Delete()`. This is in contrast to the `local_file` data source which only implements `Read()` (see figure 2.3). In this scenario I will point out when and why each of these function hooks gets called.

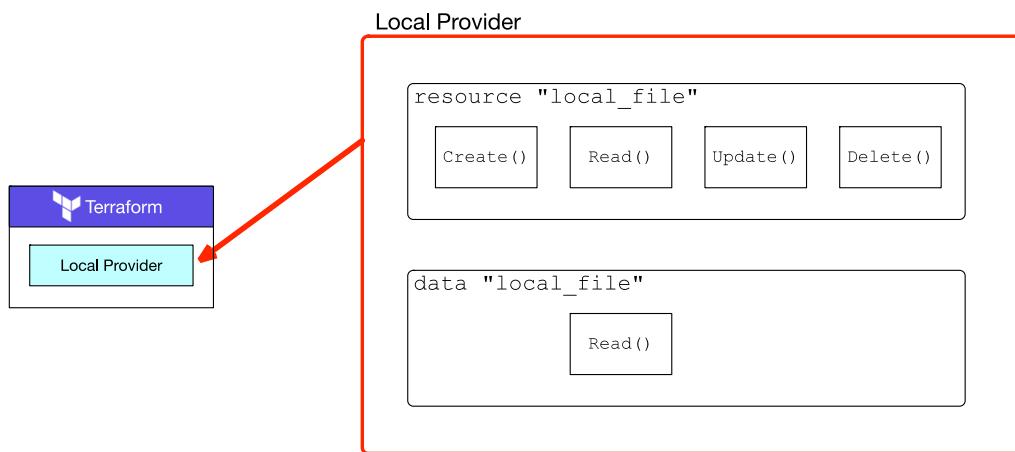


Figure 2.3 The two resources in the Local provider are a managed resource and an unmanaged data source. The managed resource implements full CRUD, while the data source only implements `Read()`

## 2.2 Declaring a Local File Resource

Let's get started by creating a new workspace for Terraform. Do this by creating a new empty directory somewhere on your computer. Make sure the folder doesn't contain any existing configuration code, because Terraform concatenates all ".tf" files together. In this workspace, make a new file called `main.tf` and add the following code from Listing 2.1.

#### Listing 2.1 Contents of main.tf

```

terraform { #A
  required_version = "~> 0.13"
  required_providers {
    local = "~> 1.4"
  }
}

```

```

resource "local_file" "literature" {
  filename = "art_of_war.txt"
  content  = <<-EOT #B
    Sun Tzu said: The art of war is of vital importance to the State.

    It is a matter of life and death, a road either to safety or to
    ruin. Hence it is a subject of inquiry which can on no account be
    neglected.
  EOT
}

```

#A Terraform settings blocks

#B Heredoc syntax for multi-line strings

**TIP** The <<- sequence indicates an indented heredoc string. Anything between the opening identifier and the closing identifier (EOT) is interpreted literally. Leading whitespace, however, is ignored (unlike traditional heredoc syntax).

There are two configuration blocks in Listing 2.1. The first block: `terraform {...}` is a special configuration block responsible for configuring Terraform itself. Its primary use is for version locking your code, but it can also configure where you state file is stored, and where providers are downloaded (we will discuss this more in chapter 6). As a reminder, the Local provider has not yet been installed yet, to do that we first need to perform `terraform init`.

The second configuration block is a resource block that declares a `local_file` resource. It provisions a text file with a given filename and content value. In this scenario, the content will contain the first couple stanzas of Sun Tzu's masterpiece, "The Art of War", and the filename will be `art_of_war.txt`. We use heredoc syntax (<<-) to input a multi-line string literal.

## 2.3 Initializing the Workspace

At this point, Terraform isn't aware of your workspace, let alone that it's supposed to create or manage anything, because it hasn't yet been initialized. Let's remedy that with a good ol' `terraform init`. `terraform init` always must be run at least once, but you'll have to run it again each time you add new provider or module dependency. Don't fret about when to run `terraform init`, because Terraform will always remind you. Moreover, `terraform init` is an *idempotent* command, which means you can call it as many times as you want in a row, and it will have no side effects.

Run `terraform init` now:

```
$ terraform init

Initializing the backend...

Initializing provider plugins...
- Finding latest version of hashicorp/local...
- Installing hashicorp/local v1.4.0...
- Installed hashicorp/local v1.4.0 (signed by HashiCorp)
```

The following providers do not have any version constraints in configuration,  
so the latest version was installed.

To prevent automatic upgrades to new major versions that may contain breaking

changes, we recommend adding version constraints in a required\_providers block in your configuration, with the constraint strings suggested below.

```
* hashicorp/local: version = "~> 1.4.0"

Terraform has been successfully initialized!
```

You may now begin working with Terraform. Try running "terraform plan" to see any changes that are required for your infrastructure. All Terraform commands should now work.

If you ever set or change modules or backend configuration for Terraform, rerun this command to reinitialize your working directory. If you forget, other commands will detect it and remind you to do so if necessary.

After initialization, Terraform creates a hidden .terraform directory for installing plugins and modules. The directory structure for the current Terraform workspace is the following:

```
.
└── main.tf
    └── .terraform
        └── plugins
            └── registry.terraform.io
                └── hashicorp
                    └── local
                        └── 1.4.0
                            └── darwin_amd64
                                └── terraform-provider-local_v1.4.0_x4
    └── selections.json
```

Because we declared a `local_file` resource in `main.tf`, Terraform is smart enough to realize that there is an implicit dependency on the Local provider, so it looks it up and downloads it from the provider registry. You don't have to declare an empty provider block ( i.e. `provider "local" {}`) unless you want to

**TIP** version lock any providers you use, whether they are implicitly or explicitly defined to ensure that any deployment you make is always repeatable.

## 2.4 Generating an Execution Plan

Before we create the `local_file` resource with a `terraform apply`, we can preview what Terraform intends to do by running `terraform plan`. You should always run `terraform plan` before deploying. I will often skip this step throughout to book for the sake of brevity, but you should still do it for yourself even if I do not call it out. `terraform plan` not only informs you what Terraform intends to do, it acts as a linter, letting you know of any syntax or dependency errors you might have. It's a read only action that does not alter the state of deployed infrastructure, and like `terraform init`, it's idempotent.

Generate an execution plan now by running `terraform plan`.

```
$ 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.
```

An execution plan has been generated and is shown below.  
 Resource actions are indicated with the following symbols:

+ create

Terraform will perform the following actions:

```
# local_file.literature will be created
+ resource "local_file" "literature" {
  + content          = <<~EOT
    Sun Tzu said: The art of war is of vital importance to the State.

    It is a matter of life and death, a road either to safety or to
    ruin. Hence it is a subject of inquiry which can on no account be
    neglected.
  EOT
  + directory_permission = "0777"
  + file_permission     = "0777"
  + filename            = "art_of_war.txt"
  + id                  = (known after apply) #A
}
```

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

Note: You didn't specify an "-out" parameter to save this plan, so Terraform can't guarantee that exactly these actions will be performed if "terraform apply" is subsequently run.

#A computed meta-attribute

### When might my plan fail?

Terraform plans can fail for many reasons, such as if your configuration code is invalid, or if there's a versioning issue, or even network related problems. Sometimes, albeit rarely, the plan will fail as a result of a bug in the provider's source code. You need to carefully read whatever error message you receive to know for sure. For more verbose logs, you can turn on trace level logging by setting the environment variable `TF_LOG` to a non-zero value, e.g. `export TF_LOG=1`

As you can see from the output, Terraform is letting us know that it wants to create a `local_file` resource. Besides the attributes that we supply, it also wants to set a computed attribute called `id`. `id` is a meta-attribute that Terraform sets on all resources. It's used to uniquely identify real-world resources, as well as internal calculations.

Although this particular `terraform` plan should have exited quickly, some plans can take a while to complete. It all has to do with how many resources you are deploying, as well as how many resources you already have in your state file.

**TIP** if terraform plan is running slow, turn off trace level logging and consider increasing parallelism (-parallelism=n)

Although the output of the plan we performed is fairly straightforward, there's a lot going on that you should be aware of. The three main stages of a terraform plan are:

1. **Reading Configuration and State** – Terraform first reads your configuration and state files (if they exist).
2. **Determine Actions to Take** – Terraform performs a calculation to determine what needs to be done to achieve your desired state. This can be one of: Create(), Read(), Update(), Delete() and No-op.
3. **Outputting Plan** – Actions need to occur in the right order, to avoid dependency problems. If we had more than one resource, this would be a bigger deal.

Figure 2.4 is a detailed flow diagram showing what happens during terraform plan.

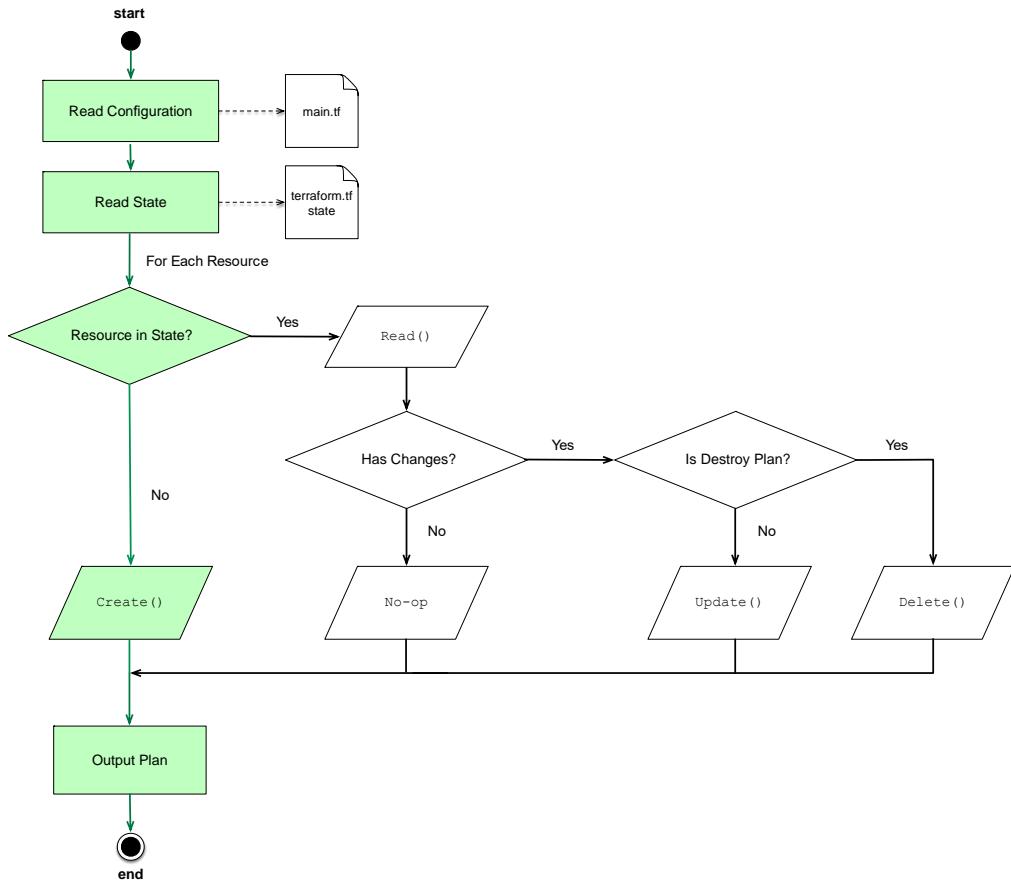
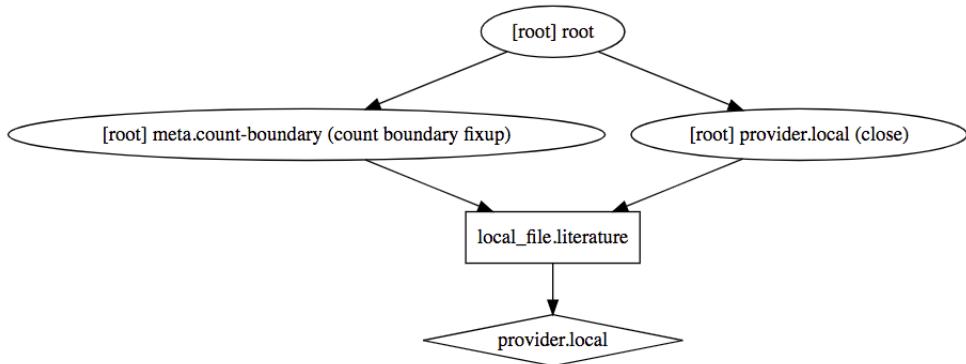


Figure 2.4 Steps that Terraform performs when generating an execution plan for a new deployment

We haven't yet talked about the dependency graph, but it's a big part of Terraform and every terraform plan generates one for respecting implicit and explicit dependencies between resource and provider nodes. Terraform has a special command for visualizing the dependency graph: `terraform graph`. This command outputs a dotfile which can be converted to a digraph using a variety of tools. You can see the resultant graph I produced for this workspace from the preceding DOT graph<sup>1</sup> in figure 2.5.

<sup>1</sup>DOT is a graph description language. DOT graphs are files with the filename extension dot. Various programs can process and render DOT files in graphical form



**Figure 2.5 Dependency graph for this workspace**

The dependency graph for this workspace has a few nodes, including one for the Local provider, one for the `local_file` resource, and a few other meta nodes that correspond to housekeeping actions. During an `apply`, Terraform will walk the dependency graph to ensure that everything is done in the right order. We'll examine a more complex digraph in the next chapter.

### 2.4.1 Inspecting the Plan

It's possible to read the output of a `terraform plan` in JSON format, which could be useful when integrating with custom tools, or enforcing Policy as Code (we will discuss Policy as Code in chapter 12).

First, save the output of the plan by setting the optional `-out` flag:

```
$ terraform plan -out plan.out
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.
```

An execution plan has been generated and is shown below.  
Resource actions are indicated with the following symbols:

+ create

Terraform will perform the following actions:

```
# local_file.literature will be created
+ resource "local_file" "literature" {
  + content          = <<~EOT
    Sun Tzu said: The art of war is of vital importance to the State.

    It is a matter of life and death, a road either to safety or to
    ruin. Hence it is a subject of inquiry which can on no account be
    neglected.
  EOT
  + directory_permission = "0777"
```

```

+ file_permission      = "0777"
+ filename             = "art_of_war.txt"
+ id                   = (known after apply)
}

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

-----
This plan was saved to: plan.out

To perform exactly these actions, run the following command to apply:
  terraform apply "plan.out"

```

plan.out is now saved as a binary file, so the next step is to convert it to JSON format. This can be done (rather unintuitively) by ingesting it with `terraform show` and piping it to an output file

```
$ terraform show -json plan.out > plan.json
```

Finally, we have the plan in human readable format:

```

$ cat plan.json
{
  "format_version": "0.1",
  "terraform_version": "0.13.0",
  "planned_values": {
    "root_module": {
      "resources": [
        {
          "address": "local_file.literature",
          "mode": "managed",
          "type": "local_file",
          "name": "literature",
          "provider_name": "registry.terraform.io/hashicorp/local",
          "schema_version": 0,
          "values": {
            "content": "Sun Tzu said: The art of war is of vital importance to the State.\n\nIt is a matter of life and death, a road either to safety or to \nruin. Hence it is a subject of inquiry which can on no account be\nneglected.\n",
            "content_base64": null,
            "directory_permission": "0777",
            "file_permission": "0777",
            "filename": "art_of_war.txt",
            "sensitive_content": null
          }
        }
      ],
      "resource_changes": [
        {
          "address": "local_file.literature",
          "mode": "managed",
          "type": "local_file",
          "name": "literature",
          "provider_name": "registry.terraform.io/hashicorp/local",
          "change": {
            "actions": ["create"],
            "before": null,
            "after": {
              "content": "Sun Tzu said: The art of war is of vital importance to the State.\n\nIt is a matter of life and death, a road either to safety or to \nruin. Hence it is a subject of inquiry which can on no account be\nneglected.\n",
              "content_base64": null,
              "directory_permission": "0777",
              "file_permission": "0777",
              "filename": "art_of_war.txt",
              "sensitive_content": null,
              "after_unknown": {
                "id": true
              }
            }
          }
        }
      ],
      "configuration": {
        "root_module": {
          "resources": [
            {
              "address": "local_file.literature",
              "mode": "managed",
              "type": "local_file",
              "name": "literature",
              "provider_config_key": "local",
              "expressions": {
                "content": {
                  "constant_value": "Sun Tzu said: The art of war is of vital importance to the State.\n\nIt is a matter of life and death, a road either to safety or to \nruin. Hence it is a subject of inquiry which can on no account be\nneglected.\n"
                },
                "filename": {
                  "constant_value": "art_of_war.txt"
                }
              },
              "schema_version": 0
            }
          ]
        }
      }
    }
  }
}

```

## 2.5 Creating the Local File Resource

Now let's run `terraform apply` to compare the output against the generated execution plan. The command and output are shown below.

```

$ terraform apply

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
+ create

Terraform will perform the following actions:

# local_file.literature will be created

```

```
+ resource "local_file" "literature" {
+   content           = <<~EOT
    Sun Tzu said: The art of war is of vital importance to the State.

    It is a matter of life and death, a road either to safety or to
    ruin. Hence it is a subject of inquiry which can on no account be
    neglected.
  EOT
+   directory_permission = "0777"
+   file_permission      = "0777"
+   filename             = "art_of_war.txt"
+   id                   = (known after apply)
}

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

Do you want to perform these actions?
Terraform will perform the actions described above.
Only 'yes' will be accepted to approve.

Enter a value:
```

Do they look similar? It's no coincidence. The execution plan generated by `terraform apply` is the exactly the same as the plan generated by `terraform plan`. In fact, you could even apply the results of `terraform plan` explicitly, i.e:

```
$ terraform plan -out plan.out && terraform apply "plan.out"
```

**TIP** separating the plan and apply like this could be useful when running Terraform in automation

Regardless of how you generate an execution plan, it's always a good idea to review the contents of the plan before applying. During an apply, Terraform creates and destroys real infrastructure, which of course has real-world consequences. If you are not careful, then a simple mistake or typo could wipe out all your entire infrastructure before you even have a chance to react.

For this workspace, there's nothing to worry about because we aren't creating "real" infrastructure anyways.

Returning to the command line, enter "yes" at the prompt to approve the manual confirmation step. Your output will be as follows.

```
$ terraform apply
...
" Enter a value: yes

local_file.literature: Creating...
local_file.literature: Creation complete after 0s [id=907b35148fa2bce6c92cba32410c25b06d24e9af]

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

Two files were created as a result of this command, `art_of_war.txt`, and `terraform.tfstate` file. You current directory (excluding hidden files) will now be:

```
.
├── art_of_war.txt
└── main.tf
```

### └─ `terraform.tfstate`

The `terraform.tfstate` file you see here is the state file that Terraform uses to keep track of the resources it manages. It's used to perform diffs during the plan and detect configuration drift. Snippet 2.1 shows what the current state file looks like.

#### Snippet 2.1 Contents of `terraform.tfstate`

```
{
  "version": 4, #A
  "terraform_version": "0.13.0",
  "serial": 1,
  "lineage": "0fe3ddc2-1d7e-1116-d554-c030de42d9ea",
  "outputs": {},
  "resources": [
    {
      "mode": "managed",
      "type": "local_file",
      "name": "literature",
      "provider": "provider[\"registry.terraform.io/hashicorp/local\"]",
      "instances": [
        {
          "schema_version": 0,
          "attributes": { #A
            "content": "Sun Tzu said: The art of war is of vital importance to the State.\n\nIt is a matter of life and death, a road either to safety or to \nruin. Hence it is a subject of inquiry which can on no account be\nneglected.\n",
            "content_base64": null,
            "directory_permission": "0777",
            "file_permission": "0777",
            "filename": "art_of_war.txt",
            "id": "907b35148fa2bce6c92cba32410c25b06d24e9af", #B
            "sensitive_content": null
          },
          "private": "bnVsbA=="
        }
      ]
    }
  ]
}
```

#A Stateful information about the `art_of_war.txt` resource  
#B The unique id of resource

**WARNING** It's important to not edit, delete or otherwise tamper with the `terraform.tfstate` file, or else Terraform could potentially lose track of the resources it manages. It is possible to restore a corrupted or missing state file, but it's difficult and time consuming to do so.

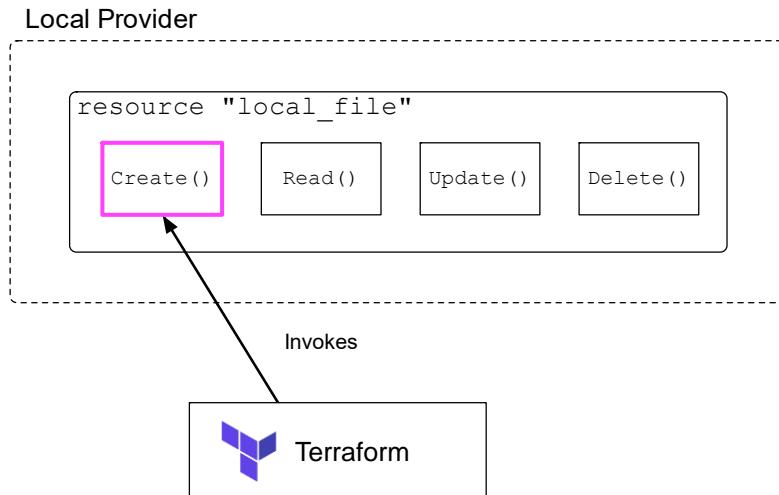
We can verify that `art_of_war.txt` matches what we expect by cat-ing the file. The command and output are shown below.

```
$ cat art_of_war.txt
Sun Tzu said: The art of war is of vital importance to the State.

It is a matter of life and death, a road either to safety or to
```

ruin. Hence it is a subject of inquiry which can on no account be neglected.

Okay, a file was created, but how did Terraform actually do that? During the apply, Terraform called `Create()` on `local_file` (see figure 2.6).



**Figure 2.6 Calling `Create()` on `local_file` during `terraform apply`**

To give you an idea of what `Create()` does, here is the actual source code from the provider:

**NOTE** relax and don't worry about understanding the code just yet. We will come back to it in chapter 10.

#### **Listing 2.2 Local File Create**

```

func resourceLocalFileCreate(d *schema.ResourceData, _ interface{}) error {
    content, err := resourceLocalFileContent(d)
    if err != nil {
        return err
    }

    destination := d.Get("filename").(string)

    destinationDir := path.Dir(destination)
    if _, err := os.Stat(destinationDir); err != nil {
        dirPerm := d.Get("directory_permission").(string)
        dirMode, _ := strconv.ParseInt(dirPerm, 8, 64)
        if err := os.MkdirAll(destinationDir, os.FileMode(dirMode)); err != nil {
            return err
        }
    }

    filePerm := d.Get("file_permission").(string)

```

```

fileMode, _ := strconv.ParseInt(filePerm, 8, 64)

if err := ioutil.WriteFile(destination, []byte(content), os.FileMode(fileMode)); err != nil
{
    return err
}

checksum := sha1.Sum([]byte(content))
d.SetId(hex.EncodeToString(checksum[:]))

return nil
}

```

## 2.6 Performing No-Op

Terraform can read existing resources to ensure they are in a desired configuration state. One way to do this is by running `terraform plan`. When `terraform plan` is run, Terraform will call `Read()` on each resource in the state file. Since our state file has only one resource, Terraform calls `Read()` on just `local_file`. Figure 2.7 shows what this looks like.

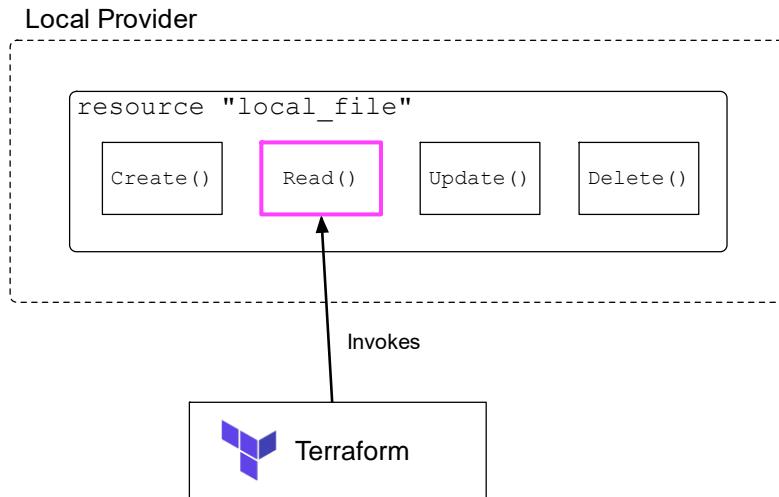


Figure 2.7 `terraform plan` calls `Read()` on the `local_file` resource

Let's run the `terraform plan` now:

```

$ 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.

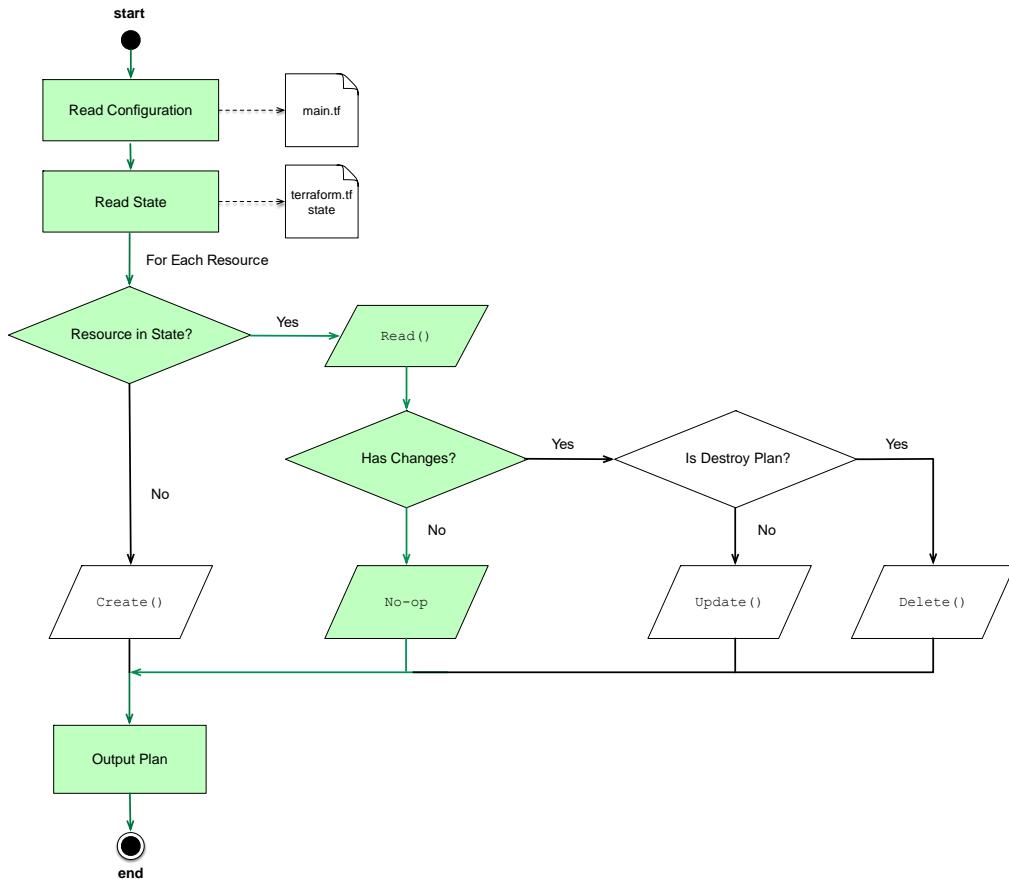
local_file.literature: Refreshing state... [id=907b35148fa2bce6c92cba32410c25b06d24e9af]

```

No changes. Infrastructure is up-to-date.

This means that Terraform did not detect any differences between your configuration and real physical resources that exist. As a result, no actions need to be performed.

There are no changes, as we would expect. The flow diagram for how a No-Op was decided is shown in figure 2.8. You can see that `Read()` is called as part of the state refresh process.



**Figure 2.8** Steps that Terraform performs when generating an execution plan for an existing deployment already in the desired state

Finally, here is the code from the provider that is performing `Read()`. Again, don't worry too much about completely understanding it.

### **Listing 2.3 Local File Read**

```

func resourceLocalFileRead(d *schema.ResourceData, _ interface{}) error {
    // If the output file doesn't exist, mark the resource for creation.
    outputPath := d.Get("filename").(string)
    if _, err := os.Stat(outputPath); os.IsNotExist(err) {
        d.SetId("")
        return nil
    }

    // Verify that the content of the destination file matches the content we
    // expect. Otherwise, the file might have been modified externally and we
    // must reconcile.
    outputContent, err := ioutil.ReadFile(outputPath)
    if err != nil {
        return err
    }

    outputChecksum := sha1.Sum([]byte(outputContent))
    if hex.EncodeToString(outputChecksum[:]) != d.Id() {
        d.SetId("")
        return nil
    }

    return nil
}

```

## 2.7 Updating the Local File Resource

You know what's better than having a file containing the first two stanzas of the "The Art of War"? That's right, having a file containing the first four stanzas of "The Art of War"! Updates are integral to Terraform and it's important to understand how they work. Update your `main.tf` code to look like Listing 2.4.

### Listing 2.4 Contents of main.tf

```

terraform {
    required_version = "~> 0.13"
    required_providers {
        local = "~> 1.4"
    }
}

resource "local_file" "literature" {
    filename = "art_of_war.txt"
    content  = <<-EOT #A
        Sun Tzu said: The art of war is of vital importance to the State.

        It is a matter of life and death, a road either to safety or to
        ruin. Hence it is a subject of inquiry which can on no account be
        neglected.

        The art of war, then, is governed by five constant factors, to be
        taken into account in one's deliberations, when seeking to
        determine the conditions obtaining in the field.

        These are: (1) The Moral Law; (2) Heaven; (3) Earth; (4) The
        Commander; (5) Method and discipline.
    EOT
}

```

```
}
```

#### #A Adding two additional stanzas

There isn't a special command for performing an update; all that needs happen is run a `terraform apply`. Before we do that though, let's run `terraform plan` to see what the generated execution plan looks like. The command and output are shown below.

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

local_file.literature: Refreshing state... [id=907b35148fa2bce6c92cba32410c25b06d24e9af]

-----
An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
-/+ destroy and then create replacement

Terraform will perform the following actions:

# local_file.literature must be replaced
-/+ resource "local_file" "literature" {
    ~ content          = <<EOT # forces replacement #B
        Sun Tzu said: The art of war is of vital importance to the State.

        It is a matter of life and death, a road either to safety or to
        ruin. Hence it is a subject of inquiry which can on no account be
        neglected.
    +
    + The art of war, then, is governed by five constant factors, to be
    + taken into account in one's deliberations, when seeking to
    + determine the conditions obtaining in the field.
    +
    + These are: (1) The Moral Law; (2) Heaven; (3) Earth; (4) The
    + Commander; (5) Method and discipline.
    EOT
    directory_permission = "0777"
    file_permission      = "0777"
    filename             = "art_of_war.txt"
    ~ id                 = "907b35148fa2bce6c92cba32410c25b06d24e9af" -> (known after apply)
}

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

-----
```

Note: You didn't specify an "-out" parameter to save this plan, so Terraform can't guarantee that exactly these actions will be performed if "`terraform apply`" is subsequently run.

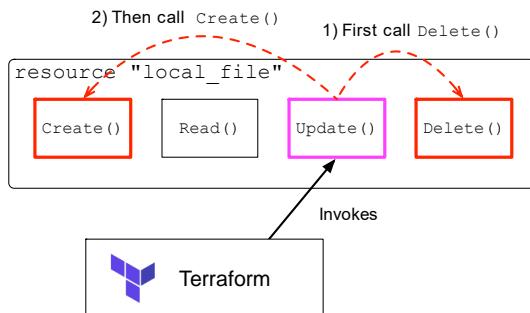
#### #A Read() happens first

#### #B Force new recreates resource

As you can see, Terraform has noticed that we've altered the `content` attribute, and is therefore proposing to destroy the old resource and create a new resource in its stead. The reason why this is done rather than update the attribute in place, is because `content` is marked as a *force new attribute*,

which means if you change it, the whole resource is tainted. In order to achieve the new desired state, Terraform must recreate the resource from scratch. This is a classic example of immutable infrastructure, although not all attributes of managed Terraform resource will behave like this. In fact, most resources have regular in-place (i.e. mutable) updates. The difference between mutable and immutable updates is shown in figure 2.9.

### Immutable Update: Force New



### Mutable Update: Normal Behavior

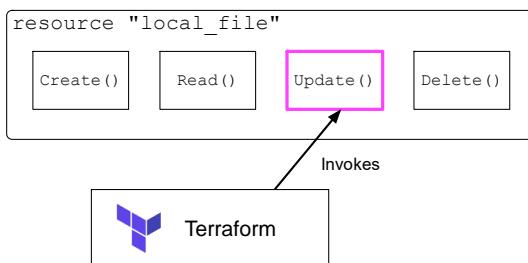
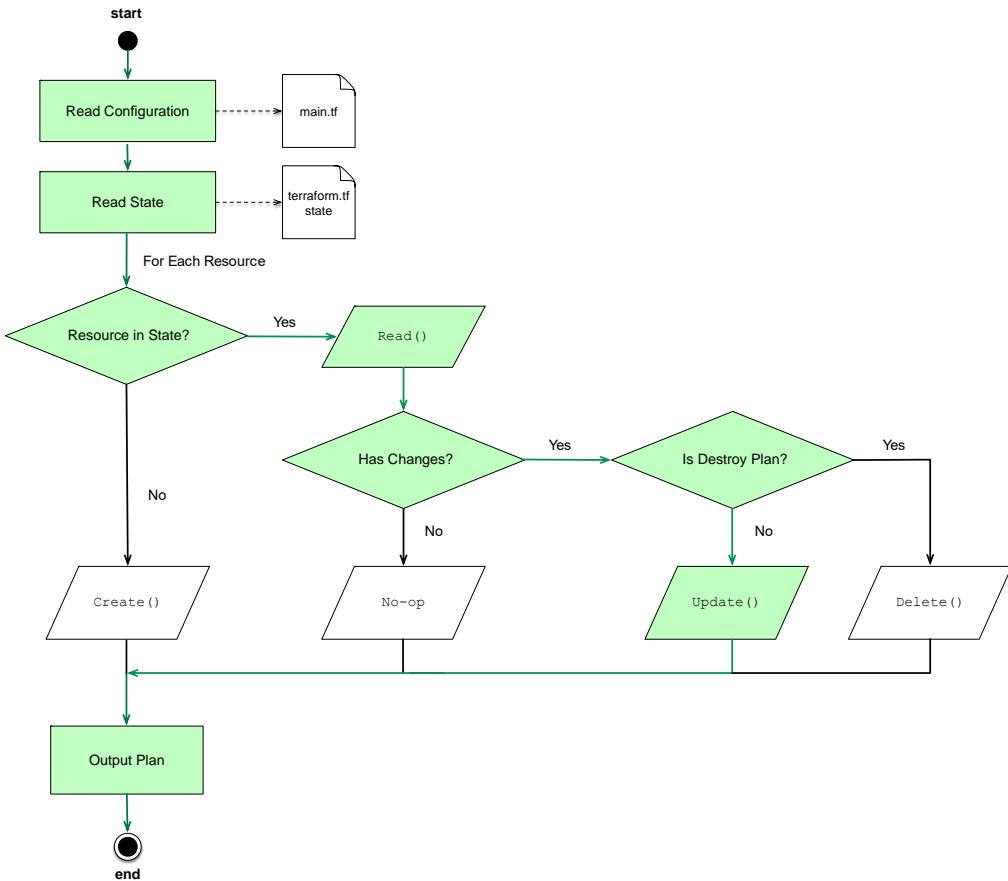


Figure 2.9 Difference between an immutable and mutable update

#### **"Force New" updates sound terrifying!**

Although destroying and recreating tainted infrastructure may sound disturbing at first, `terraform plan` will always let you know what Terraform is going to do ahead of time, so it will never come as a surprise. Furthermore, Terraform is great at creating repeatable environments, so recreating a single piece of infrastructure is not a problem. The only potential downside is if there is downtime to your service. If you absolutely cannot tolerate any downtime, then stick around for chapter 9 when we cover how to perform zero downtime deployments with Terraform.

The flow chart for the execution plan is illustrated by figure 2.10.



**Figure 2.10 Steps that Terraform performs when generating an execution plan for an update**

Go ahead and apply the proposed changes from the execution plan by running the command: `terraform apply -auto-approve`. The optional `-auto-approve` flag tells Terraform to skip the manual approval step and immediately apply changes.

**WARNING** `-auto-approve` can be dangerous if you have not already reviewed the results of the plan

```
$ terraform apply -auto-approve
local_file.literature: Refreshing state... [id=907b35148fa2bce6c92cba32410c25b06d24e9af]
local_file.literature: Destroying... [id=907b35148fa2bce6c92cba32410c25b06d24e9af]
local_file.literature: Destruction complete after 0s
local_file.literature: Creating...
local_file.literature: Creation complete after 0s [id=657f681ea1991bc54967362324b5cc9e07c06ba5]

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

You can verify that the file is now up to date by cat-ing the file once more. The command and output are as follows:

```
$ cat art_of_war.txt
Sun Tzu said: The art of war is of vital importance to the State.

It is a matter of life and death, a road either to safety or to
ruin. Hence it is a subject of inquiry which can on no account be
neglected.

The art of war, then, is governed by five constant factors, to be
taken into account in one's deliberations, when seeking to
determine the conditions obtaining in the field.

These are: (1) The Moral Law; (2) Heaven; (3) Earth; (4) The
Commander; (5) Method and discipline.
```

### 2.7.1 Detecting Configuration Drift

So far, we've been able to create and update a text file resource. But what happens if there are ad hoc changes to the file through means outside of Terraform? Configuration drift is a common occurrence in situations where there are multiple privileged users on the same file system. If you have cloud-based resources, this would be equivalent to someone making point and click changes to deployed infrastructure in the console. How does Terraform deal with configuration drift? It does so by calculating the difference between current state and desired state and performing an update.

We can simulate configuration drift by directly modifying `art_of_war.txt`. In this file, replace all occurrences of "Sun Tzu" with "Napolean".

The contents of our `art_of_war.txt` file will now be:

```
Napoleon said: The art of war is of vital importance to the
State.

It is a matter of life and death, a road either to safety or to
ruin. Hence it is a subject of inquiry which can on no account be
neglected.

The art of war, then, is governed by five constant factors, to be
taken into account in one's deliberations, when seeking to
determine the conditions obtaining in the field.

These are: (1) The Moral Law; (2) Heaven; (3) Earth; (4) The
Commander; (5) Method and discipline.
```

This misquote is patently untrue, so we'd like Terraform to detect that configuration drift has occurred and fix it. Run a `terraform plan` to see what Terraform has to say for itself.

```
$ 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.

local_file.literature: Refreshing state... [id=657f681ea1991bc54967362324b5cc9e07c06ba5]
```

An execution plan has been generated and is shown below.  
 Resource actions are indicated with the following symbols:

+ create

Terraform will perform the following actions:

```
# local_file.literature will be created
+ resource "local_file" "literature" {
  + content          = <<~EOT
    Sun Tzu said: The art of war is of vital importance to the State.

    It is a matter of life and death, a road either to safety or to
    ruin. Hence it is a subject of inquiry which can on no account be
    neglected.

    The art of war, then, is governed by five constant factors, to be
    taken into account in one's deliberations, when seeking to
    determine the conditions obtaining in the field.

    These are: (1) The Moral Law; (2) Heaven; (3) Earth; (4) The
    Commander; (5) Method and discipline.

  EOT
  + directory_permission = "0777"
  + file_permission      = "0777"
  + filename             = "art_of_war.txt"
  + id                   = (known after apply)
}
```

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

---

Note: You didn't specify an "-out" parameter to save this plan, so Terraform can't guarantee that exactly these actions will be performed if "terraform apply" is subsequently run.

Wait, what just happened here? It would appear Terraform has forgotten the resource it manages even exists and is therefore proposing to create a new resource. In fact, Terraform has not forgotten that the resource it manages still exists. The resource is still present in the state file, and you can verify by running `terraform show`:

```
$ terraform show
# local_file.literature:
resource "local_file" "literature" {
  content          = <<~EOT
    Sun Tzu said: The art of war is of vital importance to the State.

    It is a matter of life and death, a road either to safety or to
    ruin. Hence it is a subject of inquiry which can on no account be
    neglected.

    The art of war, then, is governed by five constant factors, to be
    taken into account in one's deliberations, when seeking to
    determine the conditions obtaining in the field.

    These are: (1) The Moral Law; (2) Heaven; (3) Earth; (4) The
    Commander; (5) Method and discipline.

  EOT
```

```

    directory_permission = "0777"
    file_permission      = "0777"
    filename             = "art_of_war.txt"
    id                  = "657f681ea1991bc54967362324b5cc9e07c06ba5"
}

```

The surprising outcome of `terraform plan` is merely the result of the provider choosing to do something a little odd with the way `Read()` was implemented. I don't know exactly why they chose to do it the way that they did, but they decided that if the contents of the file don't exactly match up exactly with what's in the state file, then the resource doesn't exist anymore. The consequence is that Terraform thinks that the resource no longer exists, even though there's still a file with the same name. It won't make a difference when the `apply` happens, because the existing file will get overridden anyways.

## 2.7.2 Terraform Refresh

How can we fix configuration drift? Well, Terraform automatically fixes configuration drift if you run `terraform apply`, but let's not do that right away. For now, let's have Terraform reconcile the state that it knows about with what is currently deployed. This can be done with `terraform refresh`.

You can think of `terraform refresh` like a `terraform plan` that also happens to alter the state file. It's a read-only operation that does not modify managed existing infrastructure, just Terraform state.

Returning to the command line, run a `terraform refresh` to reconcile Terraform state:

```
$ terraform refresh
local_file.literature: Refreshing state... [id=657f681ea1991bc54967362324b5cc9e07c06ba5]
```

Now if you do a `terraform show`, you can see that the state file has been updated:

```
$ terraform show
```

However, nothing is returned because this is just part of the weirdness of how the `local_file` works (it thinks the old file no longer exists). At least it is now consistent.

**NOTE** I rarely find `terraform refresh` to be useful, but some people really like it

Returning to the command line we can correct the `art_of_war.txt` file by performing a `terraform apply`:

```
$ terraform apply -auto-approve
local_file.literature: Creating...
local_file.literature: Creation complete after 0s [id=657f681ea1991bc54967362324b5cc9e07c06ba5]

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

Now the contents of `art_of_war.txt` have been restored to what it should be. If this was a cloud-based resource provisioned in AWS, GCP or Azure, any point and click changes that had been made in the console would be gone at this point. You can verify the file was successfully restored by cat-ing the file once more.

```
$ cat art_of_war.txt
Sun Tzu said: The art of war is of vital importance to the State.
```

It is a matter of life and death, a road either to safety or to ruin. Hence it is a subject of inquiry which can on no account be neglected.

The art of war, then, is governed by five constant factors, to be taken into account in one's deliberations, when seeking to determine the conditions obtaining in the field.

These are: (1) The Moral Law; (2) Heaven; (3) Earth; (4) The Commander; (5) Method and discipline.

## 2.8 Deleting the Local File Resource

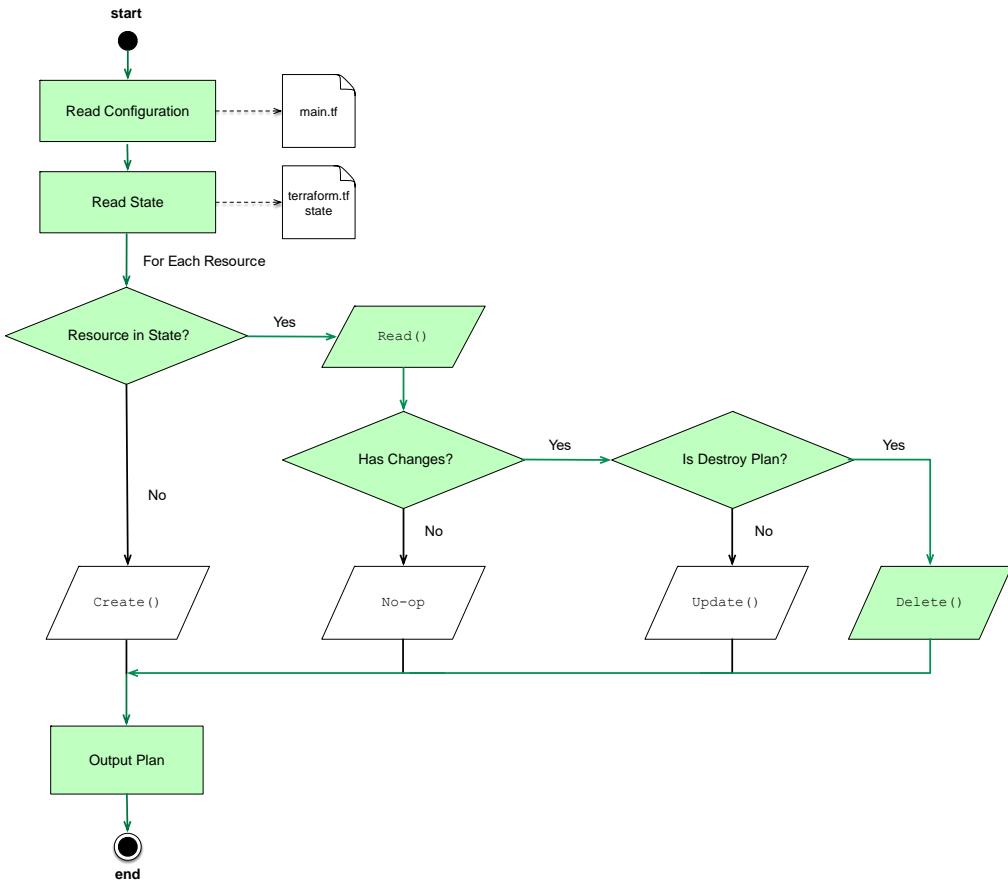
Our "Art of War" file has served us well but now it's time to say goodbye. Let's go ahead and clean-up by running a `terraform destroy`:

```
$ terraform destroy -auto-approve
local_file.literature: Refreshing state... [id=657f681ea1991bc54967362324b5cc9e07c06ba5]
local_file.literature: Destroying... [id=657f681ea1991bc54967362324b5cc9e07c06ba5]
local_file.literature: Destruction complete after 0s

Destroy complete! Resources: 1 destroyed.
```

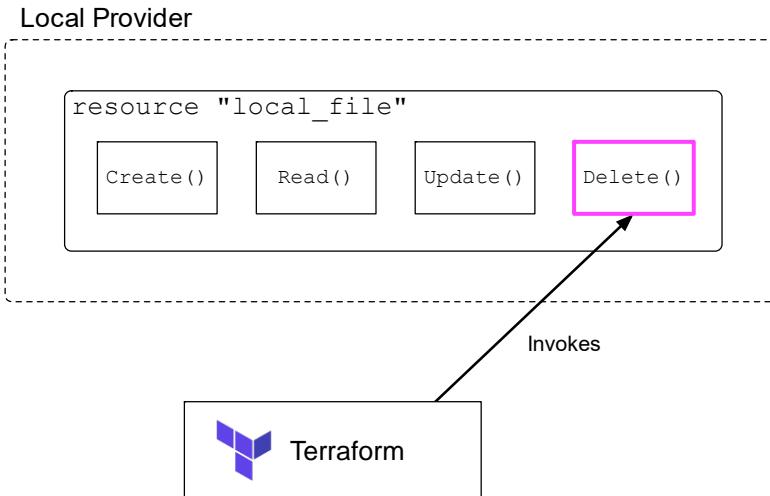
**NOTE** the optional flag `-auto-approve` for `terraform destroy` is exactly the same as for `terraform apply`; it automatically approves the result of the execution plan

The `terraform destroy` command first generates an execution plan as if there were no resources in the configuration files by performing a `Read()` on each resource and marking all existing resources for deletion. This can be seen in figure 2.11.



**Figure 2.11 Steps that Terraform performs when generating an execution plan for a delete**

During the actual execution of the destroy operation, Terraform will invoke `Delete()` on each resource in the state file. Again, since there's only one resource in the state file, Terraform effectively just calls `Delete()` on `local_file`. This is illustrated by figure 2.12.



**Figure 2.12** Terraform `destroy` calls `Delete()` on each resource in the state file

So now the `art_of_war.txt` file has been deleted. The current directory is the following:

```
.
├── main.tf
└── terraform.tfstate
└── terraform.tfstate.backup
```

**NOTE** deleting all configuration files and running `terraform apply` is equivalent to `terraform destroy`

Although it's gone, its memory lives on in a new file, `terraform.tfstate.backup`. This backup file is a copy of the previous state file and is there for purely archival purposes. This file is typically not needed and can be safely deleted if you wish, but I usually leave it be. Our current state file is empty (as far as Terraform is concerned) and is shown in snippet 2.2.

#### Snippet 2.2 Contents of `terraform.tfstate`

```
{
  "version": 4,
  "terraform_version": "0.13.0",
  "serial": 9,
  "lineage": "0fe3ddc2-1d7e-1116-d554-c030de42d9ea",
  "outputs": {},
  "resources": []
}
```

Finally, for your personal edification, here is the `Delete()` code from the Local provider (it's quite simple):

#### Snippet 2.5 Local File Delete

```
func resourceLocalFileDelete(d *schema.ResourceData, _ interface{}) error {
    os.Remove(d.Get("filename").(string))
    return nil
}
```

## 2.9 Summary

- The Local provider for Terraform allows you to create and manage text files on your machine. This is normally used to glue together “real” infrastructure but can also be useful by itself as a teaching aid.
- Resources are created in a certain sequence as dictated by the execution plan. The sequence is calculated automatically based on implicit dependencies.
- Each managed resource has lifecycle function hooks associated with them, these being: `Create()`, `Read()`, `Update()` and `Delete()`. When Terraform decides the time is right, these function hooks will be invoked.
- Changing Terraform configuration code and running a `terraform apply` will update an existing managed resource. You can also use `terraform refresh` to update the state file based on what is currently deployed.
- Terraform reads the state file during a plan to make decisions about what actions to take during an apply. It’s important not to lose the state file, or else Terraform will lose track of all the resources its managing.

# 3

## *Functional Programming*

### This chapter covers:

- Input variables, local values, and output values
- Functional programming with for expressions
- Incorporating two new providers: Random and Archive
- Templating with `templatefile()`
- Scaling resources with `count`

Functional programming is a declarative programming paradigm that allows you to do many things in a single line of code. By composing small modular functions, you are able to tell a computer *what* you want it to do, instead of *how* to do it. Functional programming is called functional programming because, as the name implies, programs consist almost entirely of functions. The core principles of functional programming are:

- **Pure Functions** – functions return the same value for the same arguments, never having any side effects.
- **First-class and Higher-order functions** – functions are treated like any other variable, and can be saved, passed around, and used to compose higher-order functions.
- **Immutability** – data is never directly modified. Instead, new data structures are created each time data would change.

To give you an idea the difference between procedural and functional programming, here is procedural JavaScript code that multiples all even numbers in an array by 10 and adds the results together:

```
const numList = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
let result = 0;
for (let i = 0; i < numList.length; i++) {
  if (numList[i] % 2 === 0) {
```

```

        result += (numList[i] * 10)
    }
}

```

And here is the same problem solved with functional programming (JavaScript):

```

const numList = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
const result = numList
    .filter(n => n % 2 === 0)
    .map(a => a * 10)
    .reduce((a, b) => a + b)

```

And in Terraform:

```

locals {
    numList = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    result  = sum([for x in local.numList : 10 * x if x % 2 == 0])
}

```

Although you may not consider yourself a programmer, it's still important to grasp the basics of functional programming. Terraform does not directly support procedural programming, so any logic you want to express needs to be declarative and functional. In this chapter we will deep dive into functions, expressions, templates, and other dynamics features that make up the Terraform language.

The specific scenario we will be looking at is building a program that generates Mad Libs from template files. Mad Libs, in case you aren't aware, is a phrasal templating word game in which one player prompts another for words to fill in the blanks of a story. An example input is shown below.

*To make a pizza, you need to take a lump of <noun>, and  
make a thin, round, <adjective> <noun>.*

For the given template string, a random noun, adjective, and another noun, will be selected to fill in the placeholders. An example output would therefore be:

*To make a pizza, you need to take a lump of roses, and  
make a thin, round, colorful jewelry.*

### 3.1 Generating a Mad Libs

Let's start by generating a single Mad Libs. To do that, we'll need a randomized pool of words to select words from, and a template file to template. The rendered content will then be printed to the CLI.

An architecture diagram for what we're about to do is shown in figure 3.1.

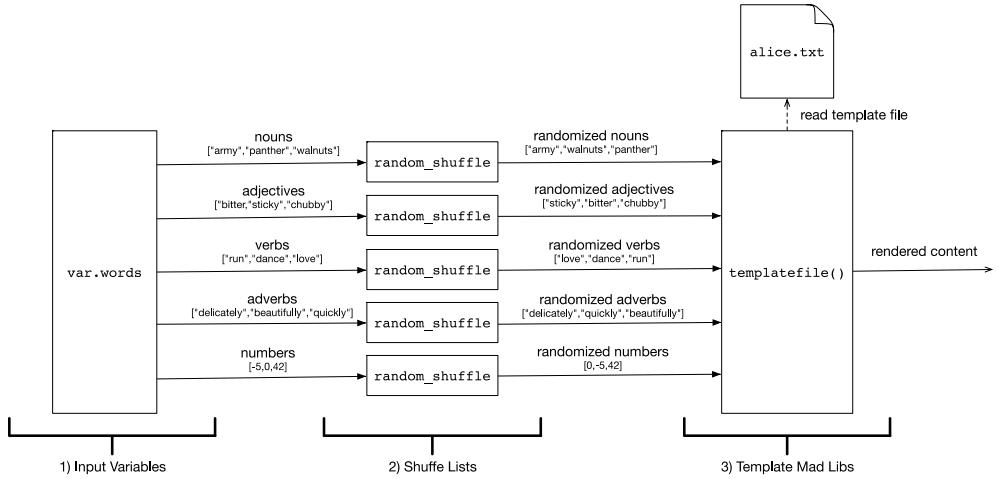


Figure 3.1 Architecture diagram of Mad Libs Template Engine

### 3.1.1 Input Variables

First, we need to create the word pool. That means we need to talk about input variables – what they are, how they are declared, and how they can be set and validated.

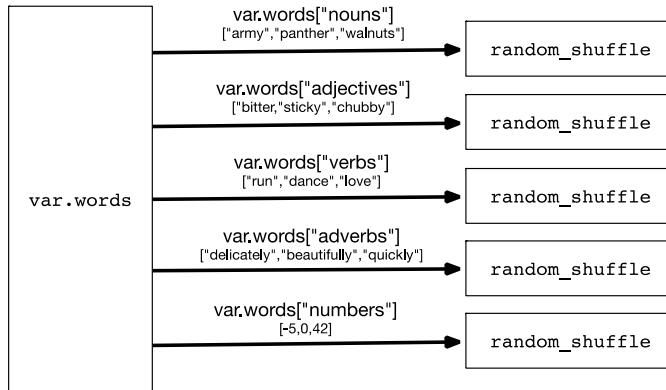
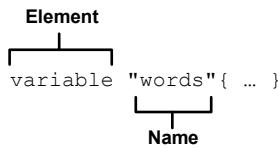


Figure 3.2 Shuffling lists of strings from `var.words`

Input variables (or Terraform variables, or just variables) are user supplied values that parametrize Terraform modules without altering the source code. Variables are declared with a variable block, which is an HCL object with two labels. The first label indicates object type, which is “variable”, and the second is the variable’s name. Variable names can be almost anything, so long as they are unique within a given module, and not a reserved identifier. The syntax of a variable block is shown in figure 3.3.



**Figure 3.3 Syntax of a variable.**

Variable blocks accept four input arguments:

- **default** – a preselected option to use when no alternative is available. Leaving this argument blank means that a variable is mandatory and must be explicitly set
- **description** – a string value providing helpful documentation to the user
- **type** – a type constraint to set for the variable. Types can be either primitive (e.g. string, integer, bool) or complex (e.g. list, set, map, object, tuple)
- **validation** – a nested block able to enforce custom validation rules

**NOTE** Variable values can be accessed within a given module by using the expression:  
var.<VARIABLE\_NAME>.

For this scenario, we could define a separate variable for each particle of speech, such as nouns, adjectives, verbs, etc. If we did that, our code would look like Snippet 3.1.

### Snippet 3.1 Multiple variable declarations

```

variable "nouns" {
  description = "A list of nouns"
  type        = list(string)
}

variable "adjectives" {
  description = "A list of adjectives"
  type        = list(string)
}

variable "verbs" {
  description = "A list of verbs"
  type        = list(string)
}

variable "adverbs" {
  description = "A list of adverbs"
  type        = list(string)
}

variable "numbers" {
  description = "A list of numbers"
  type        = list(number)
}
  
```

Although the above code is clear, we'll instead group the variables together into a single complex variable because then later it'll be possible to iterate over the words using a for expressions. Create a new project workspace for your Terraform configuration, and make a new file called `madlibs.tf`. Add in the following code from Listing 3.1.

### **Listing 3.1 madlibs.tf**

```
terraform { #A
  required_version = "~> 0.13"
}

variable "words" {
  description = "A word pool to use for Mad Libs"
  type = object({ #B
    nouns      = list(string),
    adjectives = list(string),
    verbs      = list(string),
    adverbs    = list(string),
    numbers    = list(number),
  })
}

#A Terraform settings block
#B any set value must be coercible into this complex type
```

---

### **Type Coercion – How everything you know, and love is a string**

The type of object key “numbers” in `var.words` could be `list(string)` instead of `list(number)` because of type coercion. Type coercion is the ability to convert any primitive type in Terraform to its string representation. For example, boolean true/false are converted to “true” and “false”, while numbers are similarly converted (e.g. 17 to “17”). Many people are not even aware that type coercion is a thing, because it happens so seamlessly. In fact, type coercion occurs whenever you perform string interpolation without explicitly casting the value to a string with `tostring()`. Type coercion is important to be aware of because if you accidentally coerced a value into a string, it changes the result of certain calculations (the expression `17=="17"` returns `false`, and not `true`)

---

### **3.1.2 Assigning Values with a Variable Definition File**

Assigning variable values with the `default` argument is not a good idea because it does not facilitate code reuse. A better way to set variable values is with a variables definition file, which is any file ending in either `.tfvars` or `.tfvars.json`. A variables definition file uses the same syntax as Terraform configuration code but consists exclusively of variable assignments.

Create a new file in the workspace called `terraform.tfvars` and add the following code from Listing 3.2:

### **Listing 3.2 terraform.tfvars**

```
words = {
  nouns      = ["army", "panther", "walnuts", "sandwich", "Zeus", "banana", "cat",
                "jellyfish", "jigsaw", "violin", "milk", "sun"]
  adjectives = ["bitter", "sticky", "thundering", "abundant", "chubby", "grumpy"]
```

```

verbs      = ["run", "dance", "love", "respect", "kicked", "baked"]
adverbs    = ["delicately", "beautifully", "quickly", "truthfully", "wearily"]
numbers    = [42, 27, 101, 73, -5, 0]
}

```

### 3.1.3 Validating Variables

Input variables can be validated with custom rules by declaring a nested `validation` block. To validate that at least 11 nouns are passed into `var.words`, you could write a validation block as shown in Snippet 3.2.

#### Snippet 3.2 Validation block

```

variable "words" {
  description = "A word pool to use for Mad Libs"
  type = object({
    nouns      = list(string),
    adjectives = list(string),
    verbs      = list(string),
    adverbs    = list(string),
    numbers    = list(number),
  })

  validation {
    condition    = length(var.words["nouns"]) >= 11
    error_message = "At least 11 noun must be supplied"
  }
}

```

The condition argument in `validation` is an expression that determines whether or not a variable is valid. `true` means that it's valid, while `false` means invalid. Invalid expressions will exit with an error, and the error message `error_message` will be displayed to the user. Here is an example from the user's perspective:

#### Snippet 3.3 Variable Validation Error

```

Error: Invalid value for variable

on main.tf line 5:
  5: variable "words" {

At least 11 noun must be supplied.

This was checked by the validation rule at main.tf:15,3-13.

```

**TIP** there is no limit to the number of validation blocks you can have on a variable, allowing you to be as fine grained with validation as you like

### 3.1.4 Shuffling Lists

Now that we have words in our word pool, the next step is to shuffle them. If we don't shuffle the lists, the order would be fixed, which means generating exactly the same Mad Libs on each execution. Nobody wants to read the same Mad Libs over and over again, because where is

the fun in that? You might expect there to be a function called `shuffle()` that would shuffle a generic list, but there isn't. The reason for this lacking is because Terraform strives to be a functional programming language, which means that all functions (with the exception of two) are pure functions. Pure functions return the same result for a given set of input arguments, and do not cause any additional side effects. `shuffle()` cannot be allowed because generated execution plans would be unstable, never converging on a fixed configuration.

**NOTE** `uuid()` and `timestamp()` are the only two impure Terraform functions. These are legacy functions which should be avoided whenever possible, because of their potential for introducing subtle bugs, and because they are likely to be deprecated at some point.

The Random provider for Terraform introduces a `random_shuffle` resource for safely shuffling lists, so that's what we'll use. Since we have five lists, we'll need five `random_shuffle`'s. Insert the code from Listing 3.3 into `madlibs.tf`.

### Randomness within limits

The Random provider allows for constrained randomness within Terraform configurations, and is great for generating random strings, uuids and even pet names. It's also helpful to prevent namespace collisions of Terraform resources and for generating dynamic secrets like username and database passwords. A word of caution: if you do use the Random provider for generating dynamic secrets, make sure not to hardcode a seed, and be sure to secure your state and plan files. We'll cover more about how to do this in chapter 12.

### Listing 3.3 madlibs.tf

```
terraform {
  required_version = "~> 0.13"
  required_providers {
    random = "~> 2.2"
  }
}

variable "words" {
  description = "A word pool to use for Mad Libs"
  type = object({
    nouns      = list(string),
    adjectives = list(string),
    verbs      = list(string),
    adverbs    = list(string),
    numbers    = list(number),
  })
}

resource "random_shuffle" "random_nouns" {
  input = var.words["nouns"] #A
}

resource "random_shuffle" "random_adjectives" {
  input = var.words["adjectives"]
}
```

```

resource "random_shuffle" "random_verbs" {
  input = var.words["verbs"]
}

resource "random_shuffle" "random_adverbs" {
  input = var.words["adverbs"]
}

resource "random_shuffle" "random_numbers" {
  input = var.words["numbers"]
}

```

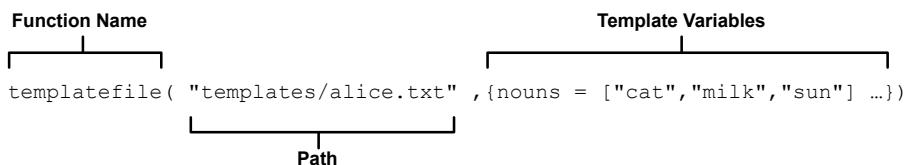
#A new shuffled list is generated from the input list

### 3.1.5 Functions

We'll use the randomized list of words to replace placeholder values in a template file, rendering content for a new Mad Libs. The built-in `templatefile()` functions allows us to easily do this. Terraform functions are expressions that transform inputs into outputs. Unlike other programming languages, Terraform does not have support for user-defined functions, nor is there a way to import functions from external libraries. Instead, you are restricted to the functions built-in to the Terraform language, which is roughly 100. That's a lot for a declarative programming language, but almost nothing compared to traditional programming languages.

**NOTE** The way you extend Terraform is by writing your own provider, not by writing new functions

Returning to the problem at hand, let's look at the `templatefile()` syntax more closely:



**Figure 3.4 Syntax of `templatefile()`**

As you can see, `templatefile()` accepts two arguments, the first being a path to the template file, and the second being a map of template variables to be rendered. We'll construct the map of template variables by aggregating the lists of shuffled words together (see figure 3.5).

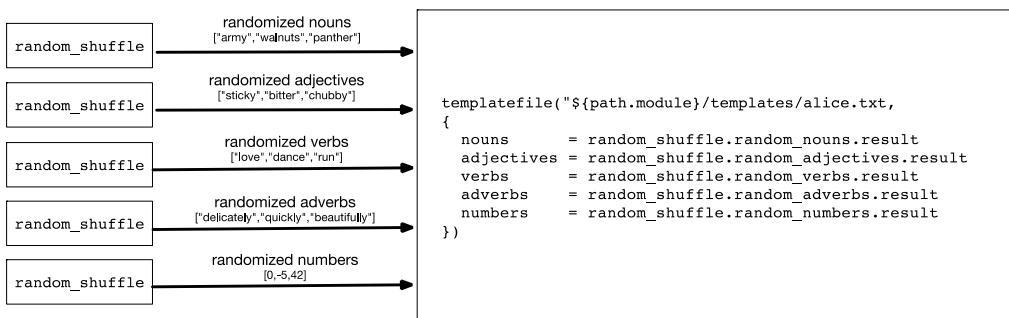


Figure 3.5 Aggregating the lists of shuffled words together into a map of template variables

Snippet 3.4 shows what the `templatefile()` code looks like:

#### Snippet 3.4 `templatefile()`

```
templatefile("${path.module}/templates/alice.txt",
{
    nouns=random_shuffle.random_nouns.result
    adjectives=random_shuffle.random_adjectives.result
    verbs=random_shuffle.random_verbs.result
    adverbs=random_shuffle.random_adverbs.result
    numbers=random_shuffle.random_numbers.result
})
```

### 3.1.6 Output Values

We can return the result of `templatefile()` back to the user with an output value. Output values are for doing one of two things:

1. Passing values between modules
2. Printing values to the CLI

We'll talk more about passing values between modules in chapter 4, for now we are interested in printing values to the CLI.

The syntax for an output block is shown in figure 3.6.

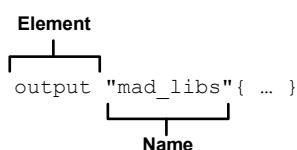


Figure 3.6 Syntax of an output value

Add the output block to `madlibs.tf`. Your configuration will now be:

**Listing 3.4 madlibs.tf**

```

terraform {
  required_version = "~> 0.13"
  required_providers {
    random = "~> 2.2"
  }
}

variable "words" {
  description = "A word pool to use for Mad Libs"
  type = object({
    nouns      = list(string),
    adjectives = list(string),
    verbs      = list(string),
    adverbs    = list(string),
    numbers    = list(number),
  })
}

resource "random_shuffle" "random_nouns" {
  input = var.words["nouns"]
}

resource "random_shuffle" "random_adjectives" {
  input = var.words["adjectives"]
}

resource "random_shuffle" "random_verbs" {
  input = var.words["verbs"]
}

resource "random_shuffle" "random_adverbs" {
  input = var.words["adverbs"]
}

resource "random_shuffle" "random_numbers" {
  input = var.words["numbers"]
}

output "mad_libs" {
  value = templatefile("${path.module}/templates/alice.txt",
  {
    nouns      = random_shuffle.random_nouns.result
    adjectives = random_shuffle.random_adjectives.result
    verbs      = random_shuffle.random_verbs.result
    adverbs    = random_shuffle.random_adverbs.result
    numbers    = random_shuffle.random_numbers.result
  })
}

```

**NOTE** `path.module` is an reference to the filesystem path of the containing module

### 3.1.7 Templates

The last thing to do is create an `alice.txt` template file. Template syntax is the same as for interpolation values in the main Terraform language, which is anything enclosed within the

markers: \${ ... }. String templates allow you to evaluate expressions and coerce the result to a string.

Any expression can be evaluated with template syntax; however, you are restricted by variable scope. Only passed in template variables are in scope, all other variables and resources – even within the same module – are not.

Let's create the template file now. First, create a new directory called `templates` to contain template files, and in this directory create an `alice.txt` file.

**TIP** Some people like to give template files a `.tpl` extension to indicate that they are template files, but I find this unhelpful and confusing. I recommend giving template files the proper extension for what they actually are.

Listing 3.5 shows the contents of `alice.txt`.

### Listing 3.5 alice.txt

ALICE'S UPSIDE-DOWN WORLD

```
Lewis Carroll's classic, "Alice's Adventures in Wonderland", as well
as its ${adjectives[0]} sequel, "Through the Looking ${nouns[0]}",
have enchanted both the young and old ${nouns[1]}s for the last
${numbers[0]} years, Alice's ${adjectives[1]} adventures begin
when she ${verbs[0]}s down a/an ${adjectives[2]} hole and lands
in a strange and topsy-turvy ${nouns[2]}. There she discovers she
can become a tall ${nouns[3]} or a small ${nouns[4]} simply by
nibbling on alternate sides of a magic ${nouns[5]}. In her travels
through Wonderland, Alice ${verbs[1]}s such remarkable
characters as the White ${nouns[6]}, the ${adjectives[3]} Hatter,
the Cheshire ${nouns[7]}, and even the Queen of ${nouns[8]}s.
Unfortunately, Alice's adventures come to a/an ${adjectives[4]}
end when Alice awakens from her ${nouns[8]}.
```

### 3.1.8 Printing Output

We're finally ready to generate our first Mad Libs. Initialize Terraform by performing a `terraform init`, then apply changes:

```
$ terraform init && terraform apply -auto-approve
...
random_shuffle.random_nouns: Creation complete after 0s [id=-]
random_shuffle.random_verbs: Creation complete after 0s [id=-]
random_shuffle.random_adverbs: Creation complete after 0s [id=-]

Apply complete! Resources: 5 added, 0 changed, 0 destroyed.

Outputs:

mad_libs = ALICE'S UPSIDE-DOWN WORLD

Lewis Carroll's classic, "Alice's Adventures in Wonderland", as well as
its abundant sequel, "Through the Looking cat",
have enchanted both the young and old Zeuss for the
last -5 years, Alice's sticky adventures begin when
```

```

she dances down a/an grumpy hole and lands
in a strange and topsy-turvy jigsaw. There she discovers she
can become a tall milk or a small jellyfish simply by
nibbling on alternate sides of a magic army. In her travels
through Wonderland, Alice runs such remarkable
characters as the White walnuts, the bitter Hatter,
the Cheshire sun, and even the Queen of panthers.
Unfortunately, Alice's adventures come to a/an thundering end
when Alice awakens from her panther.

```

## 3.2 Generating Many Mad Libs

We can generate a single Mad Libs from a randomized pool of words and output the result to the CLI. But what if we wanted to generate more than one Mad Libs? With `for` expressions and the `count` meta argument, this becomes easy.

To accomplish this, we'll need to make some changes to the original architecture. Here is the list of design changes:

1. Create 100 Mad Libs
2. Use three template files (`alice.txt`, `observatory.txt`, `photographer.txt`)
3. Capitalize each word before shuffling
4. Save Mad Libs as text files
5. Zip all Map Libs together

Our revised architecture is shown in figure 3.7.

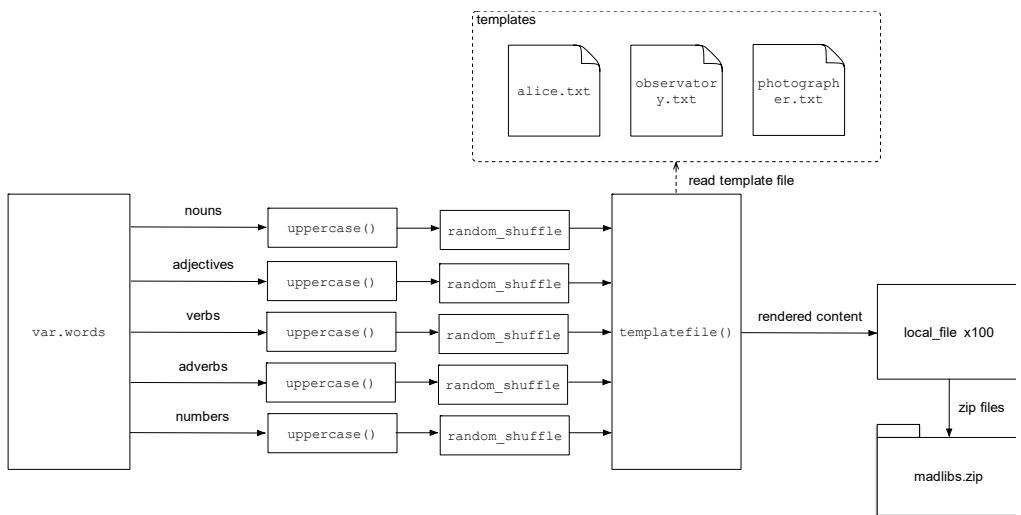


Figure 3.7 Revised architecture for Mad Libs templating engine

### 3.2.1 For Expressions

We added a step to uppercase all strings in `var.words` prior to shuffling. This isn't strictly necessary, but it does make it easier to see templated words. The result of the uppercase function is saved into a local value, which will then be fed into `random_shuffle`.

To uppercase all the strings in `var.words` we need to employ a *for expression*. For expressions are anonymous functions that can transform one complex type into another. They use lambda like syntax and are comparable to lambda expressions and streams in conventional programming languages. Figure 3.8 shows the syntax of a for expression which uppercases each element in an array of strings, and outputs the result as a new list.

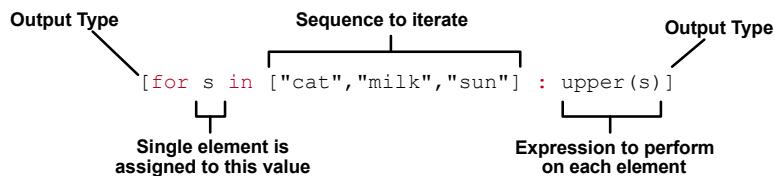


Figure 3.8 Syntax of for expression which uppercases each word in a list

A visualization of the processed stream is shown in figure 3.9.

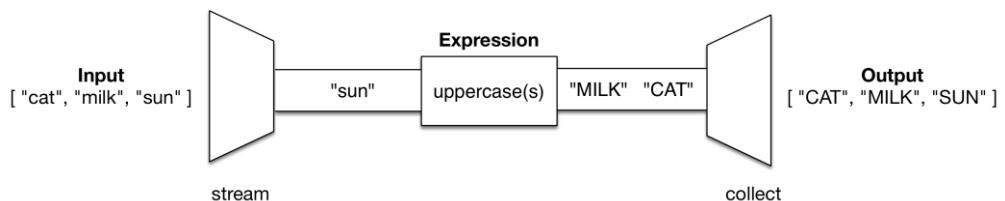


Figure 3.9 Visualization of for expression

The brackets that go around a for expression determine output type. The previous code used `[]`, which means the output will be a list. If instead we used `{}`, then the result would be an object. For example, if we wanted to loop through `var.words` and output a new map with the key being that of the original map and the value being the length of the original value, we could do that with the following expression:

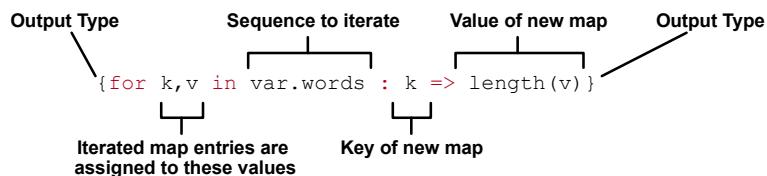


Figure 3.10 Syntax of for expression which iterates over `var.words` and outputs a map

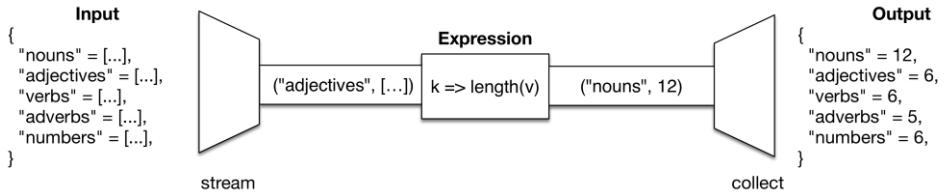


Figure 3.11 Visualization of for expression

For expressions are useful not only because they can convert one type to another, but because simple expressions can be composed to construct higher-order functions. To make a for expression which uppercases each word in `var.words`, we will compose two smaller for expressions into one *mega* for expression.

**TIP** composed for expressions hurt readability and increases cyclomatic complexity, so try not to overuse them

The general logic is as follows:

1. Loop through each key-value pair in `var.words`
2. Uppercase each word in the value list
3. Save the result to a local value

Looping through each key-value pair in `var.words`, and outputting a new map can be done with the following expression:

```
{for k,v in var.words : k => v }
```

An expression which uppercases each word in a list and outputs to a new list is:

```
[for s in v : upper(s)]
```

By combining the above two expressions we get:

```
{for k,v in var.words : k => [for s in v : upper(s)]}
```

Optionally, if you want to filter out a particular key, you can do so with the “if” clause. For example, to skip any key that matches “numbers” you could do so with the following expression:

```
{for k,v in var.words : k => [for s in v : upper(s)] if k != "numbers"}
```

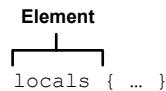
**NOTE** We do not need to skip the “numbers” key (even if it makes sense to do so) because `uppercase("1")` is equal to “1”, so it’s effectively an identity function.

### 3.2.2 Local Values

We can save the result of an expression by assigning to a local value. Local values assign a name to an expression, allowing it to be used multiple times without repetition. In making the

comparison with traditional programming languages, if input variables are analogous to a function's arguments, and output values are analogous to a function's return values, then local values are analogous to a function's local temporary symbols.

Local values are declared by creating a code block with label "locals". The syntax for a locals block is shown in figure 3.12.



**Figure 3.12 Syntax of a local value**

Add in the new local value to `madlibs.tf` and update the reference of all `random_shuffle` resources to point to `local.uppercase_words` instead of `var.words`. Listing 3.6 shows what your code should now be.

#### **Listing 3.6 madlibs.tf**

```

terraform {
  required_version = "~> 0.13"
  required_providers {
    random = "~> 2.2"
  }
}

variable "words" {
  description = "A word pool to use for Mad Libs"
  type = object({
    nouns      = list(string),
    adjectives = list(string),
    verbs      = list(string),
    adverbs    = list(string),
    numbers    = list(number),
  })
}

locals { #A
  uppercase_words = {for k, v in var.words : k => [for s in v : upper(s)]}
}

resource "random_shuffle" "random_nouns" {
  input = local.uppercase_words["nouns"]
}

resource "random_shuffle" "random_adjectives" {
  input = local.uppercase_words["adjectives"]
}

resource "random_shuffle" "random_verbs" {
  input = local.uppercase_words["verbs"]
}

resource "random_shuffle" "random_adverbs" {

```

```

    input = local.uppercase_words["adverbs"]
}

resource "random_shuffle" "random_numbers" {
    input = local.uppercase_words["numbers"]
}

#A for expression to uppercase strings and save to a local value

```

### 3.2.3 Implicit Dependencies

At this point it's important to point out that because we're using an interpolated value to set the input attribute of `random_shuffle`, a consequence is that an implicit dependency is created between the two resources. An expression or resource with an implicit dependency won't be evaluated until after the dependency is resolved. In the current workspace, the dependency diagram looks like figure 3.13.

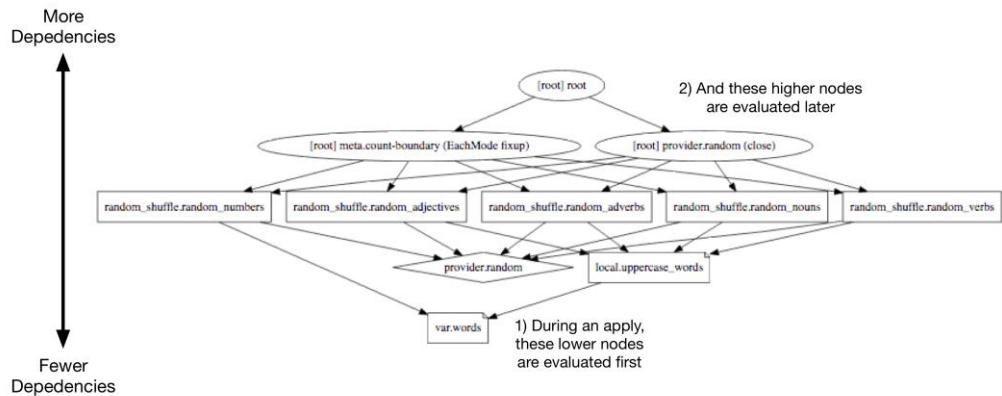


Figure 3.13 Visualizing the dependency graph and execution order

Nodes toward the bottom of the dependency graph have fewer dependencies, while nodes toward the top have more dependencies. At the very top is the root node, which is dependent on all other nodes.

What you need to know about dependency graphs are: 1) cyclical dependencies are not allowed, 2) nodes with zero dependencies are created first and destroyed last 3) you cannot guarantee any ordering between nodes at the same dependency level.

**NOTE** dependency graphs quickly become confusing when developing non-trivial projects. I do not find them useful except in the academic sense.

### 3.2.4 Count Parameter

To make 100 Mad Libs, the brute force way would be to copy our existing code 100 times and call it a day. I wouldn't recommend doing this because it's messy and doesn't scale well. Fortunately, we have better options. For this particular scenario, we'll use the count meta argument to dynamically provision resources.

**NOTE** in chapter 7 we'll cover `for_each`, which is an alternative to `count`

Count is a *meta argument*, which means all resources intrinsically support it by virtue of being a Terraform resource. The address of a managed resource is of the format: `<RESOURCE TYPE>.<NAME>`. If count is set, the value of this expression becomes a *list* of objects representing all possible resource instances. Therefore, to access the Nth instance in the list, we could do so with bracket notation, i.e. `<RESOURCE TYPE>.<NAME>[N]`.

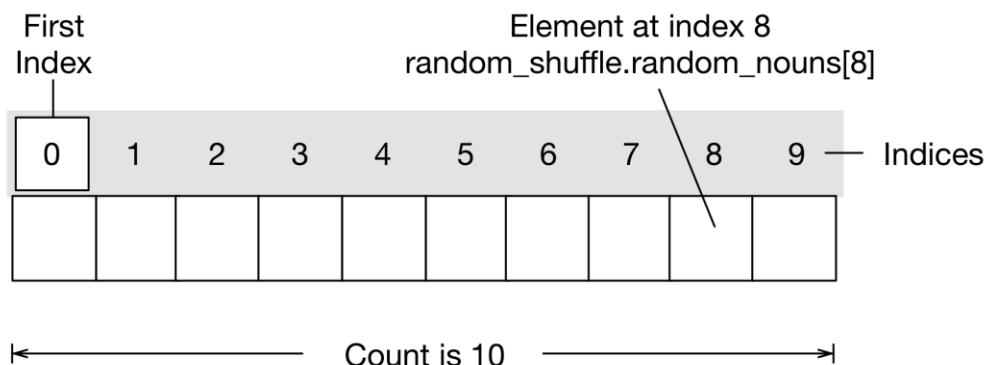


Figure 3.14 Count creates a list of resources that can be referenced using bracket notation

Let's update our code to support producing an arbitrary number of Mad Libs. First add a new variable named `var.num_files` having type number and a default value of 100. Next, reference this variable to dynamically set the count meta argument on each of the `shuffle_resources`. Your code will look like Listing 3.7.

#### Listing 3.7 madlibs.tf

```
variable "words" {
  description = "A word pool to use for Mad Libs"
  type = object({
    nouns      = list(string),
    adjectives = list(string),
    verbs      = list(string),
    adverbs    = list(string),
    numbers    = list(number),
  })
}
```

```

variable "num_files" { #A
  default = 100
  type = number
}

locals {
  uppercase_words = {for k,v in var.words : k => [for s in v : upper(s)]}
}

resource "random_shuffle" "random_nouns" {
  count = var.num_files #B
  input = local.uppercase_words["nouns"]
}

resource "random_shuffle" "random_adjectives" {
  count = var.num_files #B
  input = local.uppercase_words["adjectives"]
}

resource "random_shuffle" "random_verbs" {
  count = var.num_files #B
  input = local.uppercase_words["verbs"]
}

resource "random_shuffle" "random_adverbs" {
  count = var.num_files #B
  input = local.uppercase_words["adverbs"]
}

resource "random_shuffle" "random_numbers" {
  count = var.num_files #B
  input = local.uppercase_words["numbers"]
}

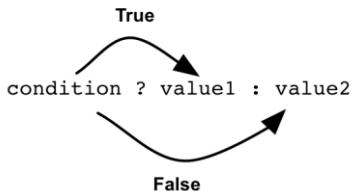
#A declaring an input variable for setting count on the random_shuffle resources
#B referencing the num_files variable to dynamically set the count meta argument

```

### 3.2.5 Conditional Expressions

Conditional expressions are ternary operators that alter control flow based on the results of a boolean condition. They can be used to selectively evaluate one of two expressions, the first for when the condition is true, and the second for when it's false. Before variables had validation blocks, conditional expressions were used to validate input variables. Nowadays, they serve a niche role.

The syntax of a conditional expression is shown in figure 3.15.



**Figure 3.15 Syntax of a conditional expression**

The following conditional expression validates that at least one noun is supplied to the nouns word list. If the condition fails, then an error will be thrown (because it is preferable to throw an error than the proceed with invalid input).

```
locals {
    v = length(var.words["nouns"])>=1 ? var.words["nouns"] : [][]#A
}
```

#A var.words["nouns"] must contain at least one word

If var.words["nouns"] contains at least one word, then application flow continues as normal. Otherwise, an error will be thrown:

**Error: Invalid index**

```
on main.tf line 8, in locals:
8:     v = length(var.words["nouns"])>=1 ? var.words["nouns"] : [][]#A
```

Lazy evaluation is why this validation trick works. Only the expression that needs to be evaluated is evaluated – the other control path is ignored. [][]#A is an expression that always throws an error if it's evaluated (since it attempts to access the 1<sup>st</sup> element of an empty list), but it's not evaluated *unless* the boolean condition is false.

More common than not, conditional expressions are used to toggle whether a resource will or will not be created. For example, if you had a boolean input variable called shuffle\_enabled, you could conditionally create a resource with the following expression:

```
count = var.shuffle_enabled ? 1 : 0
```

**WARNING** Conditional expressions hurt readability a lot, so avoid using them if you can

### 3.2.6 More Templates

Let's add two more template files to spice things up a bit. We'll cycle between them, so that we have equal number Mad Libs of each. Make a new template file called `observatory.txt` in the templates directory and set the contents to Listing 3.8:

#### **Listing 3.8 observatory.txt**

```
THE OBSERVATORY
```

```
Out class when on a field trip to a ${adjectives[0]} observatory. It
```

was located on top of a \${nouns[0]}, and it looked like a giant \${nouns[1]} with a slit down its \${nouns[2]}. We went inside and looked through a \${nouns[3]} and were able to see \${nouns[4]}s in the sky that were millions of \${nouns[5]}s away. The men and women who \${verbs[0]} in the observatory are called \${nouns[6]}s, and they are always watching for comets, eclipses, and shooting \${nouns[7]}s. An eclipse occurs when a \${nouns[8]} comes between the earth and the \${nouns[9]} and everything gets \${adjectives[1]}. Next week, we place to \${verbs[1]} the Museum of Modern \${nouns[10]}.

Afterwards, make another template file called `photographer.txt` and set the contents to Listing 3.9:

### **Listing 3.9 `photographer.txt`**

HOW TO BE A PHOTOGRAPHER

Many \${adjectives[0]} photographers make big money photographing \${nouns[0]}s and beautiful \${nouns[1]}s. They sell the prints to \${adjectives[1]} magazines or to agencies who use them in \${nouns[2]} advertisements. To be a photographer, you have to have a \${nouns[3]} camera. You also need an \${adjectives[2]} meter and filters and a special close-up \${nouns[4]}. Then you either hire professional \${nouns[1]}s or go out and snap candid pictures of ordinary \${nouns[5]}s. But if you want to have a career, you must study very \${adverbs[0]} for at least \${numbers[0]} years.

### **3.2.7 Local File**

Instead of outputting to the CLI, we'll save the results to disk with a `local_file` resource. First though, we need to read all the text files from the templates folder into a list. This is possible with the built-in `fileset()` function:

#### **Snippet 3.5 reading files in folder**

```
locals {
    templates = tolist(fileset(path.module, "templates/*.txt"))
}
```

**NOTE** sets and lists look the same but are treated as different types, so an explicit cast must be made to convert from one type to another.

Once we have the list of template files in place, we can feed the result into `local_file`. This resource will generate `var.num_files` (i.e. 100) text files:

#### **Snippet 3.6 `local_file`**

```
resource "local_file" "mad_libs" {
    count      = var.num_files
    filename  = "madlibs/madlibs-${count.index}.txt"
    content   = templatefile(element(local.templates, count.index),
    {
```

```

        nouns      = random_shuffle.random_nouns[count.index].result
        adjectives = random_shuffle.random_adjectives[count.index].result
        verbs      = random_shuffle.random_verbs[count.index].result
        adverbs    = random_shuffle.random_adverbs[count.index].result
        numbers    = random_shuffle.random_numbers[count.index].result
    })
}

```

Two things worth pointing out are `element()` and `count.index`. `element()` is a function that operates on a list as if it were a circular list, retrieving elements at a given index without throwing an out of bound exception. This means `element()` will evenly divide template files between all 100 Mad Libs.

`count.index`, is an expression that references the current index of a resource. We need it to use to parameterize filenames and ensure that `templatefile()` receives template variables from corresponding `random_shuffle` resources.

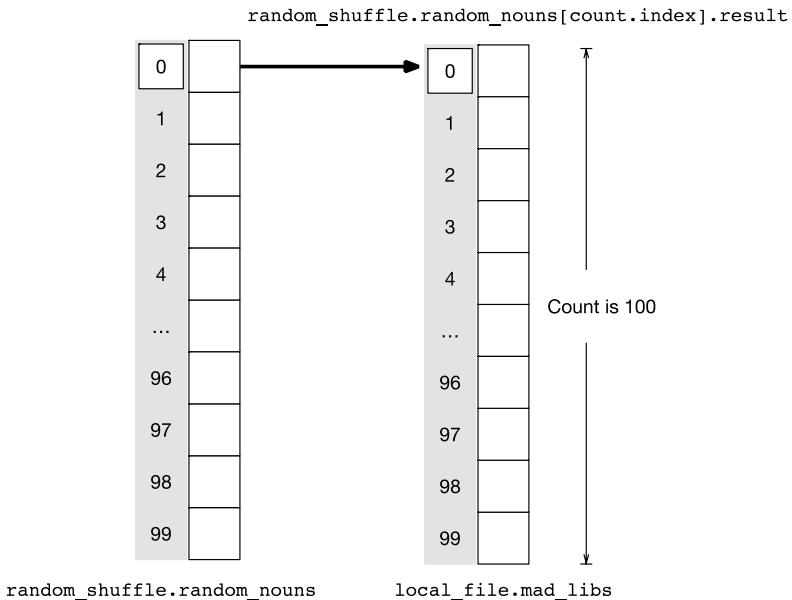


Figure 3.16 `random_nouns` and `mad_libs` are two lists of resources that must be kept in sync

### 3.2.8 Zipping Files

We have the ability to create arbitrary numbers of Mad Libs and output them in a `madlibs` directory, but wouldn't it be great to zip the files together as well? The `archive_file` data source can do just this. It takes all the files in a source directory and outputs to a new zip file. Add the following code from Snippet 3.7 into `madlibs.tf`.

**Snippet 3.7 archive\_file**

```
data "archive_file" "mad_libs" {
  depends_on = [local_file.mad_libs]
  type       = "zip"
  source_dir = "${path.module}/madlibs"
  output_path = "${path.cwd}/madlibs.zip"
}
```

`depends_on` is a meta argument that specifies explicit dependencies between resources. Explicit dependencies describe relationships between resources that are not visible to Terraform. The reason why it's included here is because `archive_file` must be evaluated after all Mad Libs have been created, otherwise it would zip up files in an empty directory. Normally we would express this relationship through an implicit dependency, which is when we use an interpolated input argument, but `archive_file` does not accept any input arguments that make sense to set from the output of `local_file`, so we are forced to use an explicit dependency instead.

**TIP** prefer implicit dependencies over explicit dependencies because they are clearer to someone reading your code. If you must use an explicit dependency, at least document the reason why you are using it and what the hidden dependency is.

For reference, the complete code for `madlibs.tf` is shown in Listing 3.10:

**Listing 3.10 madlibs.tf**

```
terraform {
  required_version = "~> 0.13"
  required_providers {
    random   = "~> 2.2"
    local     = "~> 1.4"
    archive   = "~> 1.3"
  }
}

variable "words" {
  description = "A word pool to use for Mad Libs"
  type = object({
    nouns      = list(string),
    adjectives = list(string),
    verbs      = list(string),
    adverbs    = list(string),
    numbers    = list(number),
  })
}

variable "num_files" {
  default = 100
  type    = number
}

locals {
  uppercase_words = { for k, v in var.words : k => [for s in v : upper(s)] }
}
```

```

}

resource "random_shuffle" "random_nouns" {
  count = var.num_files
  input = local.uppercase_words["nouns"]
}

resource "random_shuffle" "random_adjectives" {
  count = var.num_files
  input = local.uppercase_words["adjectives"]
}

resource "random_shuffle" "random_verbs" {
  count = var.num_files
  input = local.uppercase_words["verbs"]
}

resource "random_shuffle" "random_adverbs" {
  count = var.num_files
  input = local.uppercase_words["adverbs"]
}

resource "random_shuffle" "random_numbers" {
  count = var.num_files
  input = local.uppercase_words["numbers"]
}

locals {
  templates = tolist(fileset(path.module, "templates/*.txt"))
}

resource "local_file" "mad_libs" {
  count      = var.num_files
  filename   = "madlibs/madlibs-${count.index}.txt"
  content    = templatefile(element(local.templates, count.index),
  {
    nouns      = random_shuffle.random_nouns[count.index].result
    adjectives = random_shuffle.random_adjectives[count.index].result
    verbs      = random_shuffle.random_verbs[count.index].result
    adverbs    = random_shuffle.random_adverbs[count.index].result
    numbers    = random_shuffle.random_numbers[count.index].result
  })
}

data "archive_file" "mad_libs" {
  depends_on  = [local_file.mad_libs]
  type        = "zip"
  source_dir  = "${path.module}/madlibs"
  output_path = "${path.cwd}/madlibs.zip"
}

```

### 3.2.9 Applying Changes

We're ready to apply changes. Run a `terraform init` to download the new providers and follow it up with a `terraform apply`.

```
$ terraform init && terraform apply -auto-approve
...
```

```

local_file.mad_libs[96]: Creation complete after 0s
  [id=c9c2445f29c972aeb400831934890eb0b621747b]
local_file.mad_libs[26]: Creation complete after 0s
  [id=9d4225f7c6b2a2414ca6f8c40ff9d56e550213bd]
local_file.mad_libs[64]: Creation complete after 0s
  [id=c905d9426e9418d5be1f9d87064f90f9d5fa9d81]
local_file.mad_libs[89]: Creation complete after 0s
  [id=26e91b9456a44b74433a014fb5d587633075a9b1]

Apply complete! Resources: 600 added, 0 changed, 0 destroyed.

```

**NOTE** if you previously ran an apply before adding in archive\_file, it will say 0 resources were added/changed/destroyed. This is somewhat surprising, but it's because data sources are not considered resources for the purposes of an apply

The files in the current directory are now:

```

.
├── madlibs
│   ├── madlibs-0.txt
│   ├── madlibs-1.txt
│   ...
│   ├── madlibs-98.txt
│   └── madlibs-99.txt
└── madlibs.zip
└── madlibs.tf
└── templates
    ├── alice.txt
    ├── observatory.txt
    └── photographer.txt
└── terraform.tfstate
└── terraform.tfstate.backup
└── terraform.tfvars

```

Below is an example of a generated Mad Libs, for your personal amusement:

```

$ cat madlibs/madlibs-2.txt
HOW TO BE A PHOTOGRAPHER

Many CHUBBY photographers make big money
photographing BANANAS and beautiful JELLYFISHs. They sell
the prints to BITTER magazines or to agencies who use
them in SANDWICH advertisements. To be a photographer, you
have to have a CAT camera. You also need an
ABUNDANT meter and filters and a special close-up
WALNUTS. Then you either hire professional JELLYFISHs or go
out and snap candid pictures of ordinary PANTHERs. But if you
want to have a career, you must study very DELICATELY for at
least 27 years.

```

This is an improvement because the capitalized words stand out from the surrounding text, and, of course, because we have a lot more. To clean up, perform a terraform destroy.

**NOTE** `terraform destroy` will not delete `madlibs.zip` because this file isn't actually a managed resource. Recall that `madlibs.zip` was created with a data source, and data sources do not implement `Delete()`

### 3.3 Fireside Chat

Terraform is a highly expressive programming language. Anything you want to do is possible and rarely is the language itself an impediment. Complex logic that takes dozens of lines of procedural code can be easily expressed in one or two functional lines of Terraform code.

The focus of this chapter was on functions, expressions and templates. We started by comparing input variables, local values, and output values to the arguments, temporary symbols, and return values of a function. We then saw how we can template files using `templatefile()`.

Next, we saw how we can scale up to an arbitrary number of Mad Lib by making use of for expressions and count. For expressions allow you to compose higher order functions with lambda like syntax. This is especially useful for transforming complex data before configuring resource attributes.

The final thing we did was zip up all the Mad Libs with an `archive_file` data source. We ensured zipping was done at the right time by putting in an explicit `depends_on`.

Table 3.1 is a reference of all expressions that currently exist in Terraform.

**Table 3.1 Expression Reference**

Name	Description	Example
Conditional Expressions	Use the value of a boolean expression to select one of two values	<code>condition ? true_value : false_value</code>
Function Calls	Transform and combine values	<code>&lt;FUNCTION NAME&gt;(&lt;ARG 1&gt;, &lt;ARG2&gt;)</code>
For Expression	Transform one complex type to another	<code>[for s in var.list : upper(s)]</code>
Splat Expressions	Shorthand for some common use cases that could otherwise be handled by for expressions	<code>var.list[*].id</code> equivalent for expression: <code>[for s in var.list : s.id]</code>

<b>Dynamic Blocks</b>	<b>Construct repeatable nested blocks within resources</b>	<pre>dynamic "ingress" {   for_each = var.service_ports   content {     from_port = ingress.value     to_port = ingress.value     protocol = "tcp"   } }</pre>
<b>String Template Interpolation</b>	<b>Embed expressions in a string literal</b>	"Hello, \${var.name}!"
<b>String Template Directives</b>	<b>Use conditional results and iteration over a collection within a string literal</b>	<pre>%{ for ip in var.list.*.ip } server \${ip} %{ endfor }</pre>

### 3.4 Summary

- Input variables parameterize Terraform configurations. Local values save the results of an expression. Output values pass data around, either back to the user or to other modules.
- For expressions allow you to transform one complex type into another. They can be composed with other for expressions to create higher-order functions.
- Randomness must be constrained. Avoid using legacy functions such as `uuid()` and `timestamp()`, as these will introduce subtle bugs in Terraform due to non-convergent state.
- Zip files at runtime with the archive provider. You may need to specify an explicit dependency to ensure it runs at the right time
- `templatefile()` can template files with the same syntax used by interpolation variables. Only variables passed to this function are in scope for templating.
- The count meta argument can dynamically provision multiple instances of a resource. To access an instance of a resource created with count, use bracket notation []

# 4

## *Deploying a Multi-Tiered Web Application in AWS*

### **This chapter covers:**

- Packaging and organizing code in modules
- Best practices for structuring complex projects using modules
- Passing data between modules using input variables and output values
- Downloading and utilizing modules from the module registry
- Setting variable values from definition files

Don't panic. The code in this chapter is a little crazy, I know, especially compared to what we've done before, but it's not important that you completely understand it. Believe it or not, I actually want you to feel slightly uncomfortable, because that's an important skill in being able to navigate someone else's project structure and understand more or less how it works. The point of this chapter is not to teach you how to engineer a new project from scratch (as we'll go over how to do that in the next chapter), but how to consume modules and pass data between them. That being said, you could pretty much lift this scenario as is, and with some minor tweaks, have a production ready system.

In this chapter we're going to deploy the infrastructure for a multi-tiered web application in AWS. This is one of my favorite scenarios, because it's a fundamental problem of DevOps that Terraform neatly solves. If you work in tech, you know there's a million companies and consultants that would love to take your money by solving this problem for you. Unfortunately for them, I'm going to show you how to skip the middle man entirely and deploy all the infrastructure you need for a multi-tiered web application in only a few hundred lines of code.

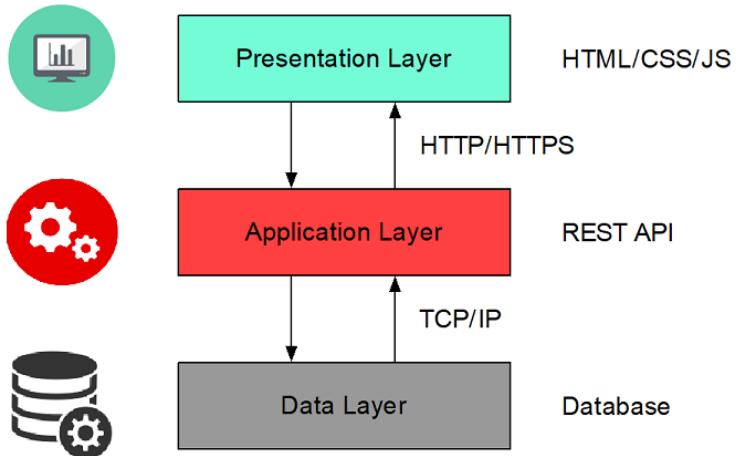


Figure 4.1 Typical Multi-Tiered Web Application

By multi-tier web application (also called N-tier), I am referring of course to a software architecture design in which a system is divided into logical “tiers” or “layers”. The 3-tier design is the most popular choice, comprising of one layer for client level logic, one layer for application (or business) level logic, and a separate data access layer for persistent, long term storage via a database or attached filesystem. Multi-tier software architecture is an industry-proven model for supporting highly available, client-server applications reliably and at scale.

At the end of the chapter, you’ll have deployed an entire solution end-to-end. The result will be a dynamic web application accessible from the internet that allows you to share pictures of your pets with your family, friends, coworkers, and the world at large. A preview of the end result is shown in figure 4.2. The multi-tiered pattern is easy to generalize, and with some minor modifications, the configuration code in this chapter could be used to host a wide assortment of web applications.

**NOTE** in chapter 5 we'll see how to deploy an analogous serverless multi-tier web application on Azure (which will be much easier to understand, believe me), and in chapters 7 and 8 we'll explore patterns for how to deploy containerized applications with Terraform.

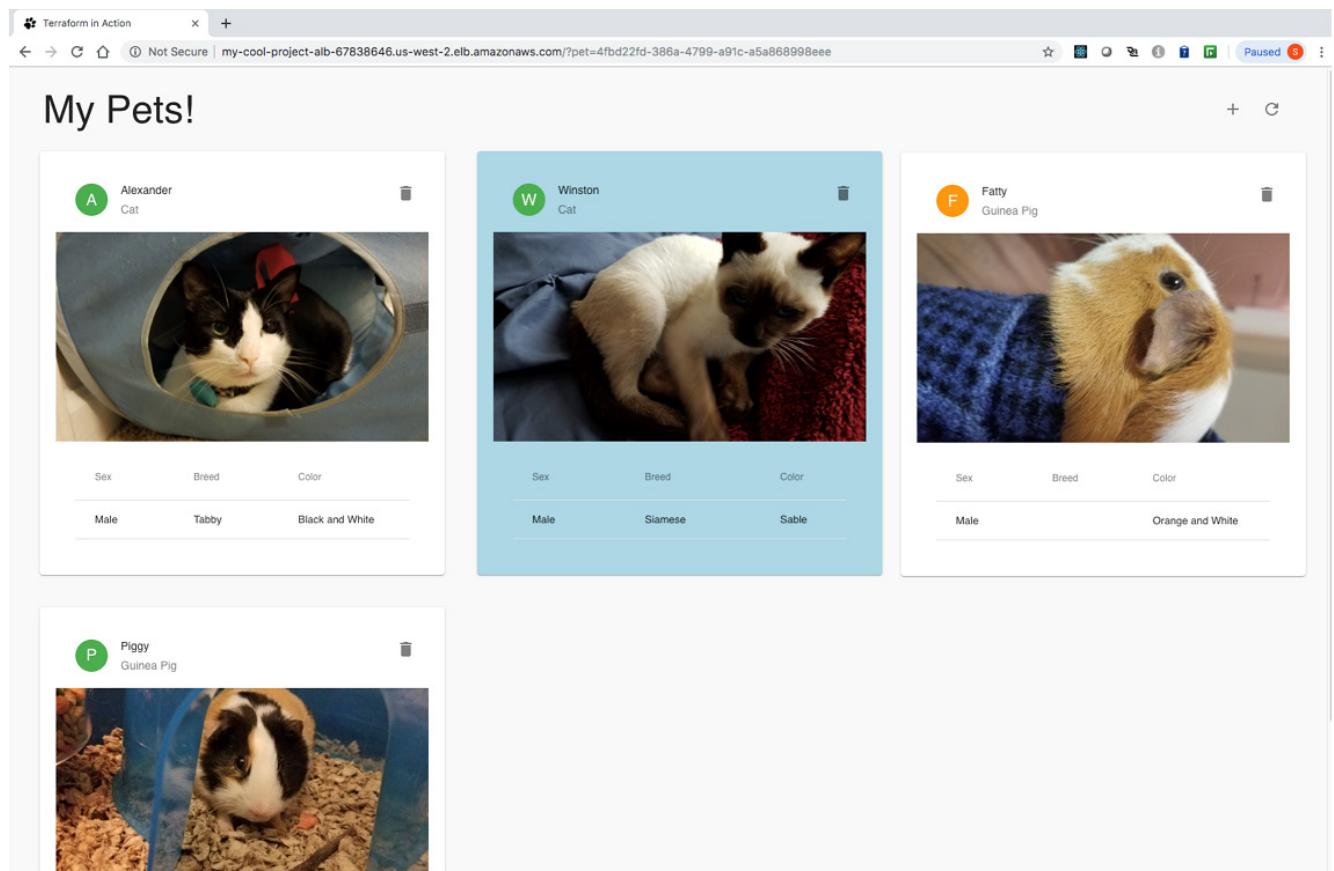


Figure 4.2 Preview of the deployed web application. Proud of my boys!

We're going to be using a lot more infrastructure in this chapter than we've used in previous chapters, including: a VPC, subnets, internet and NAT gateways, routing table and rules, load balancer, database, autoscaling group, and IAM roles. Don't fret if you are unfamiliar with all these terms! In this chapter it's more important that you understand how data flows between Terraform modules than you know everything about deploying a multi-tier web application in AWS. No matter what you choose to build with Terraform, you need to know how to use modules.

*Modules* are self-contained packages of code that allow you to create reusable components by grouping multiple related resources together. They allow you to view pieces of infrastructure only in terms of the inputs and outputs, without requiring any knowledge of the internal workings. In other words, you can treat them like little black boxes. Modules are used to for workspace organization and code reuse. As we walk through the implementation of the

web application, we'll delve into great detail how to use modules, how to pass data around, and general best practices when working with modules.

## 4.1 Motivation

Highly available and scalable web hosting can be a complex and expensive proposition. Nobody wants to go through the trouble of hosting a traditional multi-tiered web application in an on-premise data center anymore. It's simply not worth it from an economics point of view and, worse, it detracts from the core competencies of your business. This is why adopting a cloud first strategy is so critical. Major cloud providers like AWS, GCP and Azure do a great job providing reliable, scalable, secure, and high-performance infrastructure at low cost. They make it easy to port traditional multi-tiered web applications directly to the cloud with minimal alterations. Although lifting and shifting your on-premise architecture to its cloud-based equivalent is not the best strategy from a cost savings perspective (as you would be missing out the wide range of managed services offered by the cloud), it's a common approach for companies in the initial stages of migrating to the cloud. Since deploying a multi-tiered web application is a perfectly archetype problem, it's worth considering how to automate the process using Terraform.

**NOTE** It's likely your business has more heterogeneous workloads than the sleek cloud native solution covered in this chapter. If this is the case for you, then you will appreciate chapter 7 in which we cover using Terraform for the multi/hybrid cloud.

## 4.2 Architecture Overview

From a high-level perspective, we're going to use configuration files parameterized by a variables definition file to deploy the infrastructure that our web application (hereafter: webapp) needs to run. The inputs and outputs for the webapp are shown in figure 4.3.

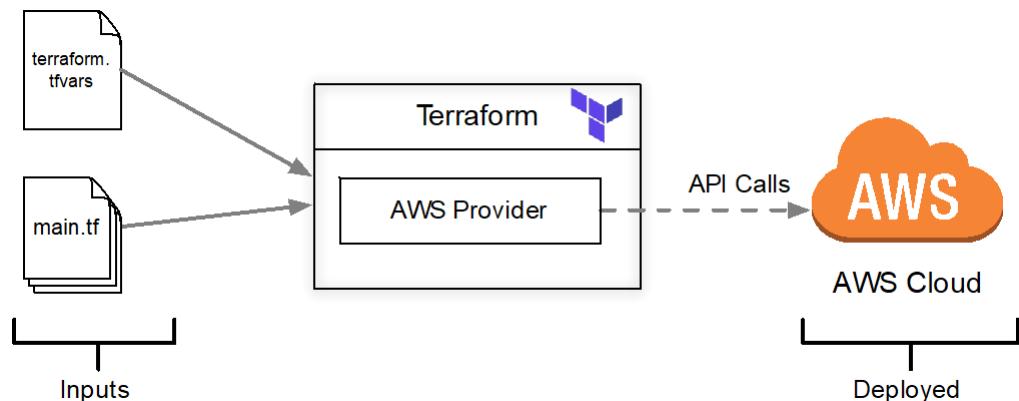


Figure 4.3 Inputs and outputs for the webapp

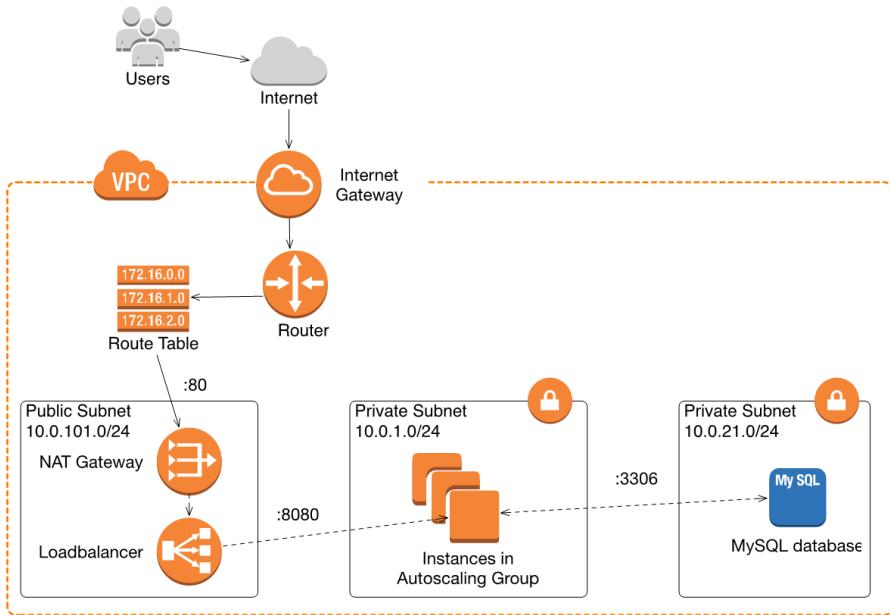
The architecture for the pet scenario is unremarkable and closely follows the design of a traditional 3-tiered web application. At a high level, clients are directed through a web portal to a service hosted on an EC2 instance, which then talks to the database layer. More specifically, traffic from the outside world will be funneled through an internet gateway to a load balancer sitting on a public subnet. That load balancer routes HTTP traffic from port 80 to EC2 instances living in an autoscaling group over port 8080. We'll shield the instances from malicious users by isolating them on a private subnet not accessible to the outside world except through a NAT gateway. Furthermore, we restrict allowable traffic to the instances using a security rule that only allows the load balancer to communicate over port 8080.

If you are unfamiliar with autoscaling groups and load balancers, I'll give you a quick rundown. An autoscaling group is just a collection of AWS EC2 instances that are treated as a logical grouping for the purposes of autoscaling and management. They allow you to automatically scale the number of instances to meet demand based on customizable scaling policies and health checks. An autoscaling group is typically put behind a load balancer to route incoming traffic dynamically across the instances, as well as allowing it to expose a single public IP address to the outside world. Instances in an autoscaling group use a launch template to start up, which you can think of as equivalent to a blueprint for individual EC2 instances.

After initialization, EC2 instances in the autoscaling group are responsible for serving both static HTML/CSS/JS content and the RESTful API. A more robust design might have the frontend layer hosted on a separate autoscaling group, and then communicate with an application layer through an internal load balancer, but that seemed unnecessarily complex for this little scenario. The infrastructure is already ridiculously scalable and powerful for the tiny application it'll be running; it's like a Volkswagen beetle with a huge Ferrari engine dropped in – a bit overkill to say the least.

The application layer persists stateful information to the data layer, which is made up of a MySQL database. The database is, like the EC2 instances, protected from the outside world because it lives in its own private subnet. It also has a security rule which allows ingress traffic only from the instances in the autoscaling group. All of the subnets are part of the same virtual private cloud (VPC) which effectively isolates the resources in a virtual network. The complete architecture diagram for what we'll be building is shown in figure 4.4

**NOTE** We're not going to cover configuring SSL or DNS on the load balancer, although it is possible to do this with Terraform. AWS allows you to create SSL certificates for free with Amazon Certificate Manager (ACM), and Route53 can be used for DNS resolution; both of these services have corresponding resources in the AWS provider.



**Figure 4.4** Architecture diagram for the multi-tiered web application

Here's how we'll do it:

1. Use modules to structure and organize the project
2. Configure and write boilerplate code for the root module
3. Write code for each of the three major modules: *networking*, *database*, and *autoscaling*, making sure data is passed properly between each
4. Deploy and verify that the website is functional

### 4.3 Modules are Your Friend

Architecting Terraform projects is a lot like playing with Lego bricks. You have the ability to combine different pieces and components together in virtually infinite ways. Like Legos, you could assemble your own creation by connecting individual blocks together to make highly customizable builds, or you could use a pre-packaged set that already has everything you need. Modules in Terraform are analogous to using pre-packaged Lego sets. They allow you to incorporate off-the-shelf infrastructure packages directly into your project, all you need to do is follow the instructions (i.e. documentation) how to use them.

Modules are a means of organizing Terraform code into discrete and reusable components. They allow you to treat groupings of resources and data sources as a single unit and can be versioned controlled to ensure immutability. Like resources and data sources, modules have inputs and outputs, and if you understand the inputs and outputs, you can reason how the

system behaves as a whole. The beauty of modules is that you don't have to know how they work to be able to use them. Modules can be endlessly nested within other modules, so they make for excellent tools for breaking complexity into small, reusable bits. Nevertheless, you typically don't want to have more than three or four levels of nested modules, otherwise it becomes difficult to reason about (like a deeply nested class hierarchy). Finally, modules can be sourced from either a local directory or from remote locations such as a generic Git or HTTP location, and the Terraform module registry. If you choose to source a module from a remote location, Terraform will download and install it for you during a `terraform init` or `terraform get`.

The syntax for a module that we'll see later in this chapter is showcased in Snippet 4.1.

#### **Snippet 4.1 syntax of module**

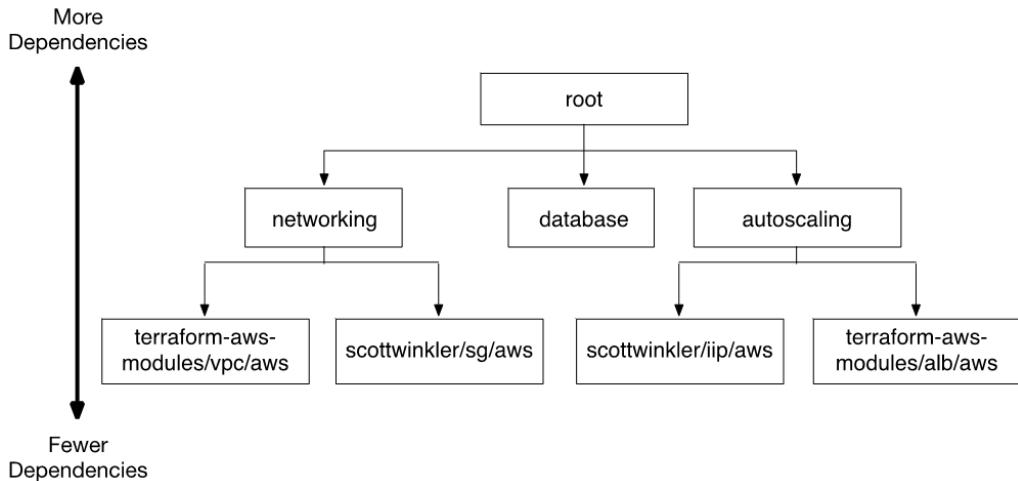
```
module "lb_sg" { #A
  source = "scottwinkler/sg/aws" #B
  version = "1.0.0" #C

  vpc_id = module.vpc.vpc_id #D
  ingress_rules = [{ #D
    port      = 80 #D
    cidr_blocks = ["0.0.0.0/0"] #D
  }] #D
}
```

#A module declaration  
#B address in Terraform module registry  
#C optional version lock  
#D input variables to module

#### **4.3.1 Standard Module Structure**

We'll follow the standard module structure for organizing our project. At the top of the hierarchy is the root module, which is the main entry point for Terraform, and where `terraform apply` is run. This root module contains a nested module for each of the three major components of the web application, namely: networking, database and autoscaling. The networking and autoscaling modules will, in turn, source their own nested submodules from remote locations. Figure 4.5 illustrates the overall design.



**Figure 4.5** Dependency tree of nested Terraform modules branching off the root module

In accordance with established code conventions and best practices from HashiCorp, we'll split module code into three unambiguously named configuration files:

- **main.tf** – the primary entry point containing all resources and data sources
- **outputs.tf** – declarations for all output values
- **variables.tf** – declarations for all input variables

For the root module we'll also create another configuration file called *providers.tf*, to contain declarations for all the providers the project utilizes. Besides the four configuration files, we'll have a *versions.tf* and variables definition file *terraform.tfvars*, for a total of six files. The variables definition file is used to set the variables in the root module, and thereby configure all of the nested modules.

In the autoscaling module there is another file besides the three configuration files, a template file called *cloud\_config.yaml*. This template file is used for the init (or user\_data) script of the EC2 instances. A more detailed structure for the project with all the files that we'll create is illustrated in figure 4.6.

**NOTE** modules sourced from remote locations are downloaded automatically, so we won't be creating files for them.

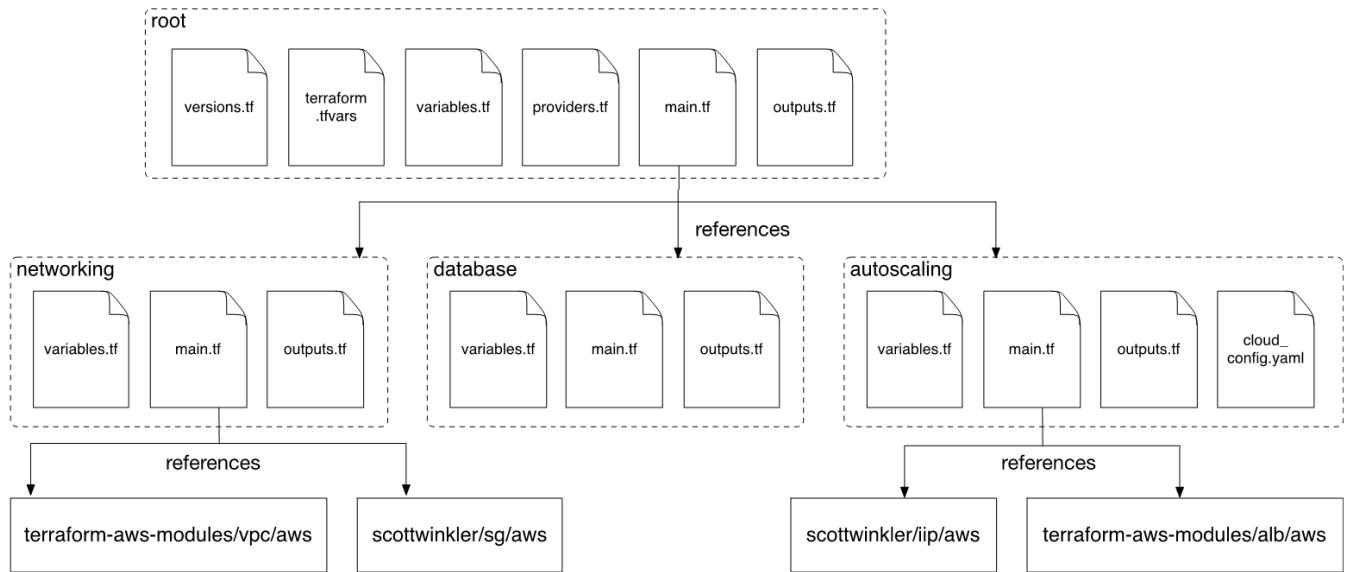


Figure 4.6 Detailed project file structure

## 4.4 Root Module

Technically, we've been using root modules all along. A root module is just the entry point for Terraform, so wherever you initialize and apply Terraform, that directory is the de-facto root module. For this project, the root module is responsible for setting all variables using a variables definition file, declaring and configuring the AWS provider, and configuring and passing data between all nested modules. We'll be using the root module mostly for code organization and won't use it to actually deploy any resources as that's the job of the nested modules. A user of the root module only needs to set the namespace variable in order to deploy the whole project. The output they'll get afterwards will contain the provisioned load balancer's DNS name (`lb_dns_name`) and the password of the database (`db_password`). The load balancer DNS name is important because it's how the user will navigate to the website from a web browser. Overall inputs and outputs of the root module are shown in figure 4.7.

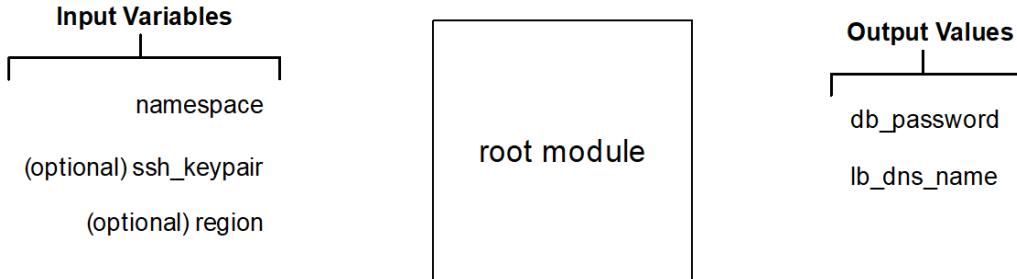


Figure 4.7 Inputs variables and output values for root module

Because this is the root module, we'll need to create six files: three that are standard for all modules, and three more that are used for additional configuration. These are:

- **variables.tf** – declarations for all input variables
- **main.tf** – declarations for all nested modules
- **outputs.tf** – declarations for all output values
- **terraform.tfvars** – the variables definition file
- **providers.tf** – explicit declaration for the AWS provider
- **versions.tf** – Terraform settings and provider version locking

Start by creating a new directory to house the project. In this folder, create a `variables.tf` file and copy in the code from listing 4.1.

#### **Listing 4.1 variables.tf**

```

variable "namespace" { #A
  description = "The project namespace to use for unique resource naming"
  type = string
}

variable "ssh_keypair" {
  description = "optional ssh keypair to use for EC2 instance"
  default     = null #B
  type        = string
}

variable "region" {
  description = "AWS region"
  default     = "us-west-2"
  type        = string
}

```

#A in many open source modules namespace is sometimes called “environment”

#B The null type are useful for variables where you either want the value set, but not empty

Next, we'll inject variable values using a variables definition file. The variables definition file allows you to parameterize configuration code without having to hardcode a default value. It uses the same basic syntax as Terraform configuration files but consists only of variable name

assignments. Create a new file named `terraform.tfvars` and insert the code from Listing 4.2 to set the variables in `variables.tf`. I have chosen not to set the `ssh_keypair` variable, but you could do so here, if you wish. It would be practical if you wanted to SSH into the machine via a bastion host.

#### **Listing 4.2 terraform.tfvars**

```
namespace = "my-cool-project"
region = "us-west-2"
```

The variables set by `terraform.tfvars` are used to configure code in `main.tf` and `providers.tf`. Let's configure the AWS provider now. Do this by creating a new file called `providers.tf` and copying in the code from Listing 4.3.

#### **Listing 4.3 providers.tf**

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

Next, create a `versions.tf` file and set the contents to that of Listing 4.4. This file will be used to lock the provider and terraform versions.

#### **Listing 4.4 versions.tf**

```
terraform {
  required_version = "~> 0.12"
  required_providers {
    local      = "~> 1.2"
    aws        = "~> 2.13"
    random     = "~> 2.1"
    template   = "~> 2.1"
  }
}
```

Finally, create a `main.tf`, `outputs.tf`. These two files will be blank for now, but we'll be coming back to them soon.

## **4.5 Networking Module**

Now that we most of the boilerplate for the root module out of the way, it's time to start work on the first chunk of code which actually does something interesting: the networking module. This module is responsible for provisioning all of the networking components for the webapp to function. These include: the VPC, public and private subnets, NAT Gateway, internet gateway, route table, route table associations, and security groups. The inputs and outputs for the networking module are shown in figure 4.8.

**NOTE** some of the resources provisioned by the networking module are not covered under the AWS free tier

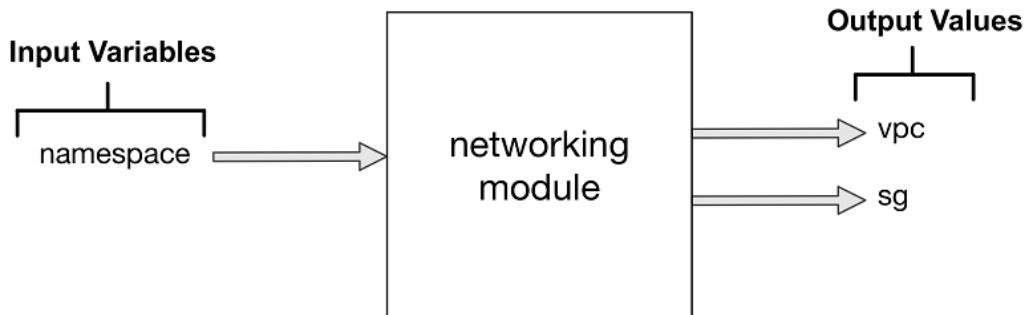


Figure 4.8 inputs and outputs of the networking module

Aside from the overall I/O, the networking module has the side effect or provisioning managed resources for all the networking components in AWS. If it didn't do this, then it would be pretty useless, wouldn't it? Resources are, of course, provisioned during a `terraform apply`. This is illustrated by figure 4.9.

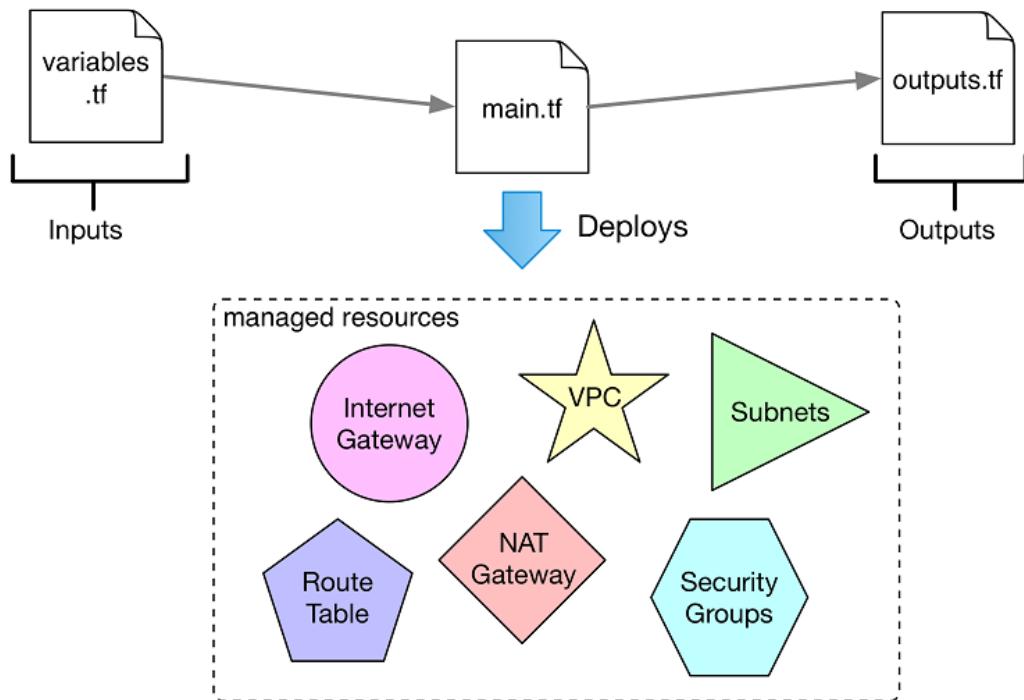


Figure 4.9 managed resources provisioned by the networking module

Create a new directory in the current working directory: `./modules/networking`. In this directory, create three new files: `variables.tf`, `main.tf`, and `outputs.tf`. The code for `variables.tf` file is blessedly short, since the only input is namespace:

#### **Listing 4.5 variables.tf**

```
variable "namespace" {
    type = string
}
```

The code doing the real work is seen in Listing 4.6. If you are already familiar with Terraform, you might disagree with the resources I have chosen to group together in the networking module. That's fine, because it's a matter of personal preference and there is no "right" way to structure these things. Some people like to group resources based purely on service, (for example: EC2, RDS, Route53, VPC, etc.), but I like to group resources based on functionality.

**NOTE** relax and remember that 100% understanding of the code isn't all that important. Pay attention instead to how the data flows. There's some good graphics coming up that'll make this much clearer, so hang with me for a little while longer.

#### **Listing 4.6 main.tf**

```
data "aws_availability_zones" "available" {}#A

module "vpc" {
    source                  = "terraform-aws-modules/vpc/aws" #B
    version                = "2.5.0"
    name                   = "${var.namespace}-vpc"
    cidr                   = "10.0.0.0/16"
    azs                     = data.aws_availability_zones.available.names
    private_subnets         = ["10.0.1.0/24", "10.0.2.0/24", "10.0.3.0/24"]
    public_subnets          = ["10.0.101.0/24", "10.0.102.0/24", "10.0.103.0/24"]
    database_subnets        = ["10.0.21.0/24", "10.0.22.0/24", "10.0.23.0/24"]
    assign_generated_ipv6_cidr_block = true
    create_database_subnet_group = true
    enable_nat_gateway       = true
    single_nat_gateway       = true
}

module "lb_sg" {
    source = "scottwinkler/sg/aws" #C
    vpc_id = module.vpc.vpc_id
    ingress_rules = [
        {
            port      = 80
            cidr_blocks = ["0.0.0.0/0"]
        }
    ]
}

module "websvr_sg" {
    source = "scottwinkler/sg/aws" #C
    vpc_id = module.vpc.vpc_id
    ingress_rules = [
        {
            port      = 8080
        }
    ]
}
```

```

        security_groups = [module.lb_sg.security_group.id]
    },
{
    port      = 22 #D
    cidr_blocks = ["10.0.0.0/16"] #D
}
]
}

module "db_sg" {
    source = "scottwinkler/sg/aws" #C
    vpc_id = module.vpc.vpc_id
    ingress_rules = [
        {
            port      = 3306
            security_groups = [module.websvr_sg.security_group.id]
        }
    ]
}

```

#A Data source which lists all the availability zones in the current region  
#B Source for the VPC module published in the Terraform Module Registry by AWS  
#C Source for a module I published for creating security groups  
#D Allow SSH for potential bastion host

**NOTE** nearly the entire networking module is made up of other nested modules. This is known as *software componentization*, which is the practice of breaking up large and complex software systems into smaller, easier to identify subsystems that have well-defined interfaces.

## Build vs. Buy

Although I could show you how to write the Terraform configuration code for deploying all the subsystems of the networking components in nitty gritty detail, I won't because A) it's not necessary, and B) it would be long and boring. AWS has conveniently published a module called `terraform-aws-vpc` which does all the hard VPC provisioning stuff for us. I'll go over in greater detail naming conventions and publishing techniques of modules in chapter 6. The VPC module is one case where it makes sense to use an off the shelf solution rather than building out a custom module from individual resource blocks, but I've still taken the precaution of locking in the version to avoid any potentially breaking changes in the future. For my own modules, I tend not lock in a version, because I always want to fetch the latest version (which is what happens by default if you do not specify a version).

You should always evaluate for yourself whether or not it's worth the tradeoff to use an external module vs. writing the Terraform code yourself. Using other people's code can save you time in the short term, but it can also be a source of tech debt later down the road, especially if something were to break in an unexpected way. Relying on open source modules is inherently risky, as there could be backdoors, unmaintained code, or the repo could simply be deleted without notice. Forking the repo and/or locking in a version could solve this problem, but it's all about who you trust. I only trust external modules with a large community backing them, because that way I feel safe knowing there's many people monitoring and updating the source code. Even then, I always skim through the module to see what it's doing before incorporating into the rest of my configuration code.

The code for `outputs.tf` is shown in Listing 4.7. Notice that the VPC output passes a reference to the entire output of the VPC module. This allows us to be succinct in the outputs code, especially when passing data through multiple layers of nested modules. Also notice that

the sg output is made up of a new object containing the ids of the security groups. This pattern is useful for grouping related attributes from different resources together into a single output value.

#### **Listing 4.7 outputs.tf**

```
output "vpc" {
  value = module.vpc #A
}

output "sg" {
  value = { #B
    lb      = module.lb_sg.security_group.id #B
    db      = module.db_sg.security_group.id #B
    websvr = module.websvr_sg.security_group.id #B
  } #B
}
```

#A passing a reference to the entire vpc module as an output  
#B constructing a new object containing the id for each of the three security groups

Finally, update `main.tf` in the root module to include a reference to the networking module. You can now (optionally) test that everything is working by running a `terraform init` and `terraform plan` from the root module.

#### **Listing 4.8 main.tf in root module**

```
module "networking" {
  source   = "./modules/networking" #A
  namespace = var.namespace #B
}
```

#A Relative path to the networking module  
#B Input variable passed from the root module

## **4.6 Database Module**

The database module is the simplest and shortest of the three major modules. Its purpose in life is to provision the database and that's it. Although the database module barely warrants having its own module, it's still an independent system in its own right, so we'll humor it. The inputs and outputs of the database module are shown in figure 4.10.

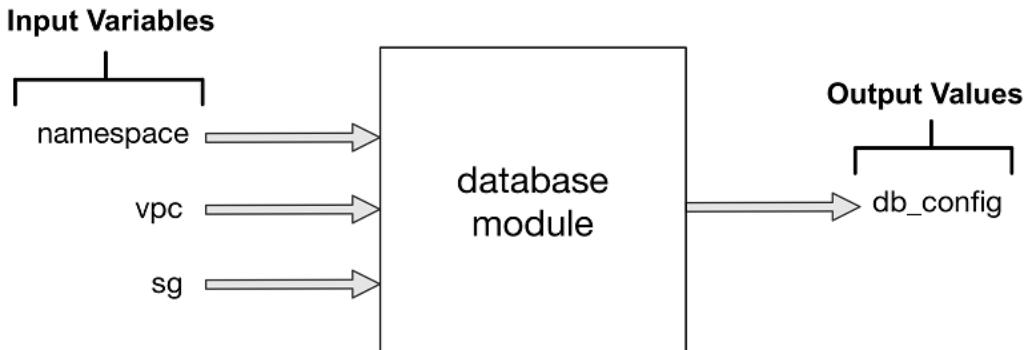


Figure 4.10 inputs and outputs of the database module

In addition to the namespace variable, the database module requires two other input variables: vpc and sg. Both of these inputs will come from the output of the networking module. The database module is therefore implicitly dependent on the networking module and won't be evaluated until after the outputs of the networking module are available. The output of the module is the configuration of the database which will be an object constructed similarly to what we did for sg in the previous section.

As I mentioned earlier, there's only one managed resource created by this module, so the side effect diagram is dirt simple compared to what we saw for the networking module. This is illustrated by figure 4.11.

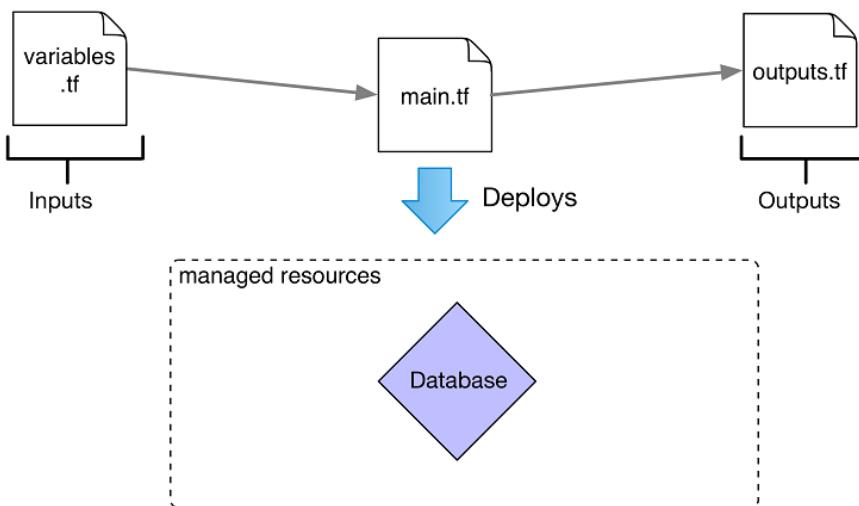


Figure 4.11 managed resources provisioned by the database module

### 4.6.1 Passing Data from the Networking Module

As part of the input configuration of the database module, we need some data from the networking module, specifically the security group id and the database subnet group name. All data being passed between modules is either in the process of “bubbling up” or “trickling down”. This is analogous to how event handling works. Data produced by a nested module will pass through its parent modules, as high as necessary until it is appropriately handled and can start trickling down to where it needs to go. For example, since the database module needs to know what its security group id is, that piece of information will be bubbled up from the scottwinkler/sg/aws module, passed though the networking module, then trickled down into the database module. This can be visualized by figure 4.12.

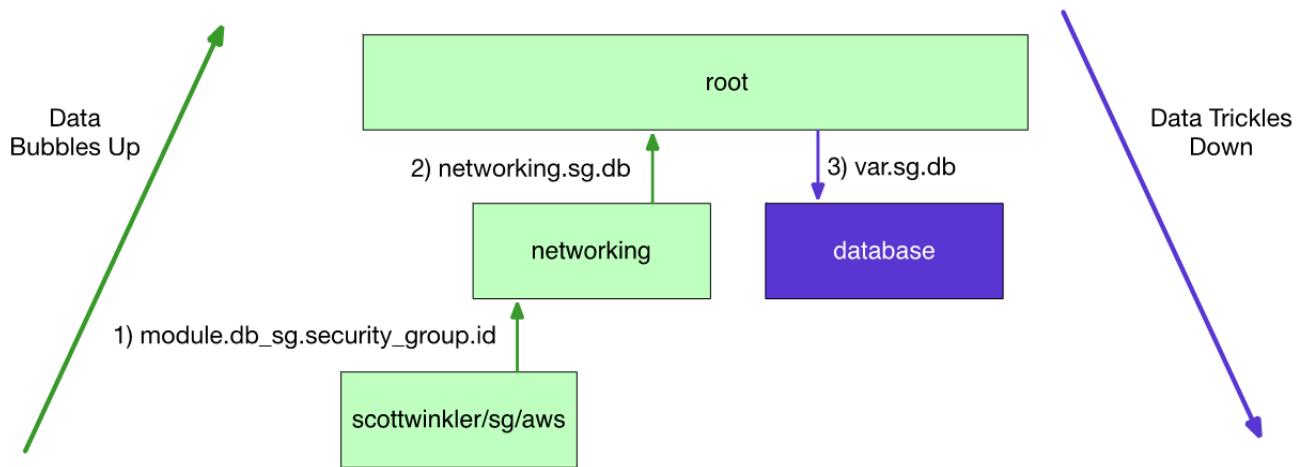


Figure 4.12 Data flow for how the database’s security group id makes its way from the scottwinkler/sg/aws module to the database module

**TIP** reasoning about how data needs to pass between modules will often make it clear how you should componentize your software systems. Modules that need to share a lot of data should be closer together, while modules that are more independent can be further apart.

Since we have already done (1) and (2) from figure 4.12, all we need to do now is trickle the data down into the database module by passing it as input variables. Listing 4.9 shows how to do this. Remember that besides the security group id, we also need to pass in the database subnet name. I’ve included this now for the sake of brevity.

#### Listing 4.9 main.tf in root module

```
module "database" {
  source    = "./modules/database"
  namespace = var.namespace
```

```

vpc = module.networking.vpc #A
sg  = module.networking.sg #A
}

module "networking" {
  source    = "./modules/networking"
  namespace = var.namespace
}

```

#A data bubbled up from the networking module and trickling down from the root module

For (3) we need to declare input variables in the database module. Create a new directory under `./modules` called “database” and in this directory, create three files: `variables.tf`, `main.tf` and `outputs.tf`. The contents of your `variables.tf` file will contain the input variables for `namespace`, `vpc` and `sg`.

#### **Listing 4.10 variables.tf**

```

variable "namespace" {
  type = string
}

variable "vpc" {
  type = any #A
}

variable "sg" {
  type = any #A
}

```

#A A type constraint of “any” type means that Terraform will skip type checking

**WARNING** while it may be tempting to overuse the `any` type constraint, it’s a lazy coding habit that will get you in trouble more often than not. Only use `any` when passing data between modules, and never for configuring the variables of the root module.

#### **4.6.2 Generating a Random Password**

Now that we have the data in the module scope, we can reference it in the configuration code as normal. The code for `main.tf` of the database module is shown in Listing 4.11. I know I said we were only going to create one resource, but I thought you might appreciate a surprise visit from our old friend, the random provider. We’ll use the `random_id` resource of the random provider to generate a cryptographic password for the database.

#### **Listing 4.11 main.tf**

```

resource "random_id" "random_16" {
  byte_length = 16 * 3 / 4 #A
}

locals {

```

```

db_password = random_id.random_16.b64_url #B
}

resource "aws_db_instance" "database" { #C
  allocated_storage      = 10
  engine                 = "mysql"
  engine_version         = "5.7"
  instance_class          = "db.t2.micro"
  identifier              = "${var.namespace}-db-instance"
  name                   = "pets"
  username                = "admin"
  password                = local.db_password
  skip_final_snapshot     = true
  db_subnet_group_name    = var.vpc.database_subnet_group #D
  vpc_security_group_ids = [var.sg.db] #D
}

```

#A using the random provider to create a 16 characters long string

#B setting a local value to an output attribute of random\_16

#C an AWS managed database

#D these values came from the networking module

**NOTE** the  $\frac{3}{4}$  factor in random\_id has to do with a golang transformation between bytes and characters

Next, we construct an output value consisting of all the crucial database configuration information (Listing 4.12). This is done in a similar fashion to what we did with the sg object in the networking module. In this situation, however, instead of aggregating data from multiple resources into one, the purpose of this object is to bubble up just the minimum amount of data that the other modules need to function. This is in accordance with the *principle of least privilege*.

#### Listing 4.12 outputs.tf

```

output "db_config" {
  value = {
    user      = aws_db_instance.database.username #A
    password  = aws_db_instance.database.password #A
    database  = aws_db_instance.database.name #A
    hostname  = aws_db_instance.database.address #A
    port      = aws_db_instance.database.port #A
  }
}

```

#A all of the data in db\_config comes from select output of the aws\_db\_instance resource

**TIP** never grant more access to data than a given module needs for legitimate purposes. As in normal software development, the inputs and outputs of a module represent the interface, so don't pass in more data than required by the interface.

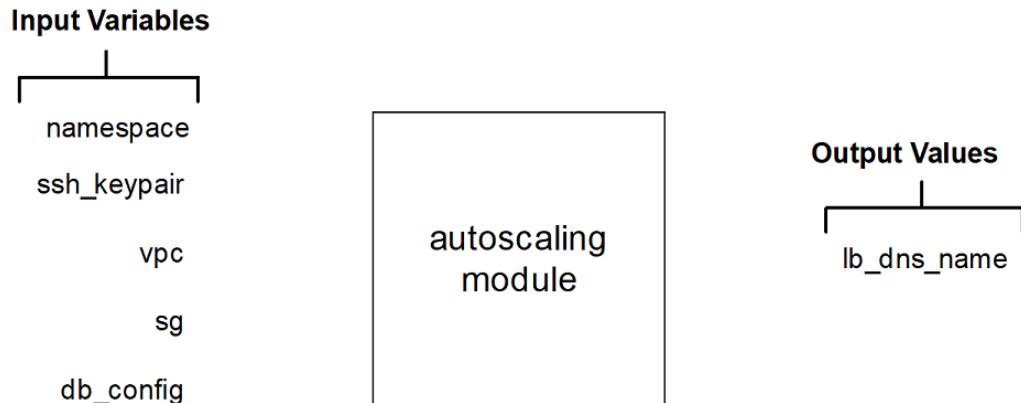
Finally, make the database password available to the user by adding an output value to in outputs.tf of the root module. Only outputs from the root module appear in the CLI after a terraform apply.

#### **Listing 4.13 outputs.tf in root module**

```
output "db_password" {
  value = module.database.db_config.password
}
```

## **4.7 Autoscaling Module**

Believe it or not, we're getting close to the end. This is the last major module, but fortunately/unfortunately it's also the most complex. This module is responsible for provisioning the autoscaling group, load balancer, IAM instance role, and everything else that the web server needs to run and serve up a healthy application. The inputs and outputs of the autoscaling module are illustrated by figure 4.13.



**Figure 4.13 inputs and outputs of the autoscaling module**

Like the networking module, the autoscaling module will be provisioning a lot of resources. These are depicted by figure 4.14.

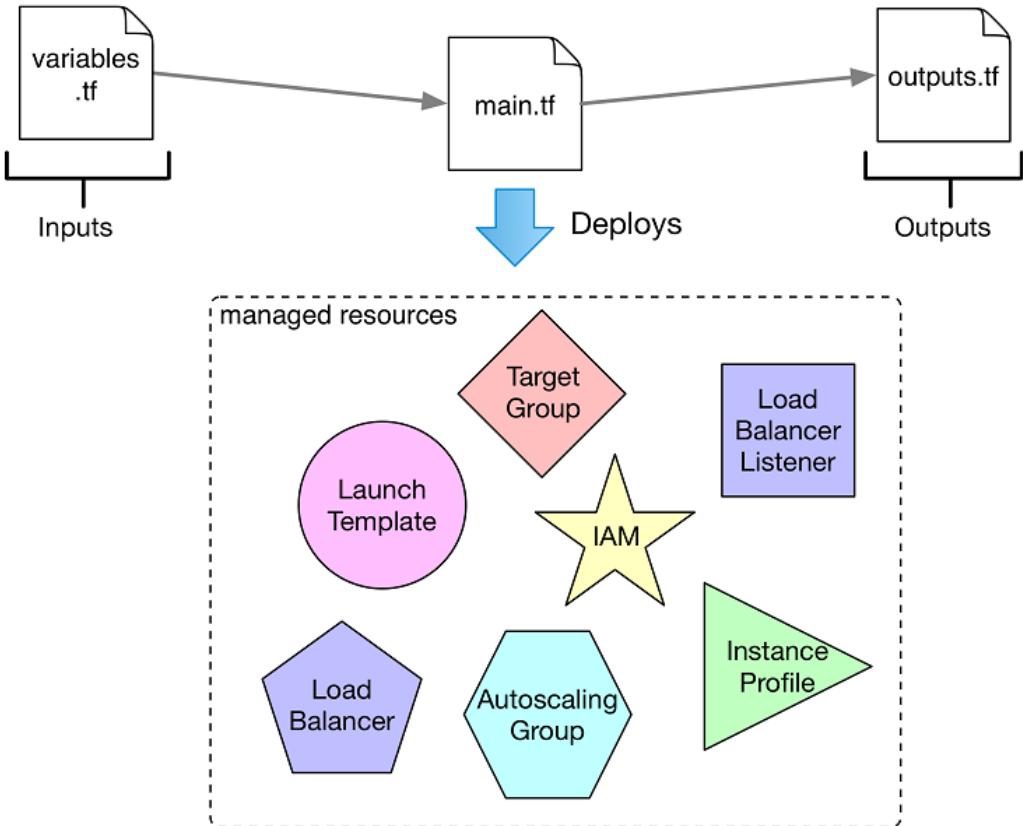


Figure 4.14 managed resources provisioned by the autoscaling module

#### 4.7.1 Trickling Down Data

From figure 4.15, it's clear we need to inject three additional variable values: `vpc`, `sg` and `db_config`. The first two will come from the networking module, same as before, while the last comes from the database module. The way data bubbles up from the networking module and trickles down into the ALB module is shown in figure 4.15. I'm only going to show this one, but the other data values are passed similarly.

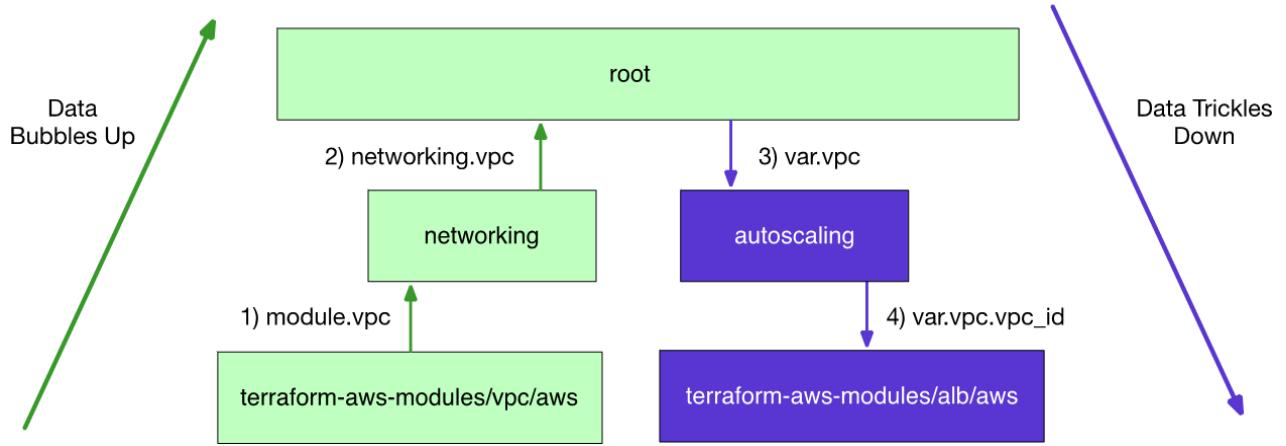


Figure 4.15 Data flow for how vpc id makes its way from the `terraform-aws-vpc` to the `terraform-aws-alb` module

The revised code for `main.tf` in the root module is shown in Listing 4.14.

#### **Listing 4.14 main.tf in root module**

```

module "autoscaling" {
  source      = "./modules/autoscaling"
  namespace   = var.namespace
  ssh_keypair = var.ssh_keypair

  vpc        = module.networking.vpc #A
  sg         = module.networking.sg #A
  db_config  = module.database.db_config #A
}

module "database" {
  source      = "./modules/database"
  namespace   = var.namespace

  vpc = module.networking.vpc
  sg  = module.networking.sg
}

module "networking" {
  source      = "./modules/networking"
  namespace   = var.namespace
}

```

#A input arguments for the autoscaling module, set by other module's outputs

As before, the input variables of the module will be used to set the variables in `variables.tf`. First, create an `autoscaling` directory under `./modules`, then put in four new files: `main.tf`, `variables.tf`, `outputs.tf` and `cloud_config.yaml`. The last one is a template file. Template files don't need to end in ".txt", so I generally use an extension that makes it clearer

what the template file actually is. Listing 4.15 presents the code for `variables.tf`, in the autoscaling module.

#### **Listing 4.15 variables.tf**

```
variable "namespace" {
  type = string
}

variable "ssh_keypair" {
  type = string
}

variable "vpc" {
  type = any
}

variable "sg" {
  type = any
}

variable "db_config" {
  type = object( #A
    { #A
      user      = string #A
      password  = string #A
      database  = string #A
      hostname  = string #A
      port      = string #A
    } #A
  ) #A
}
```

#A Enforcing a strict type schema for the db\_config object. The value set for the variable must implement the type schema

#### **4.7.2 Detailed Module Planning**

The infrastructure in this module is trickier than the other two modules we've seen and requires a bit of explanation. As a recap, we're going to be using an autoscaling group behind a load balancer, with a launch template for startup configuration. I like to draw a rough diagram of the dependencies between resources and modules before I start writing any code. An initial dependency diagram I came up with is depicted in figure 4.16.

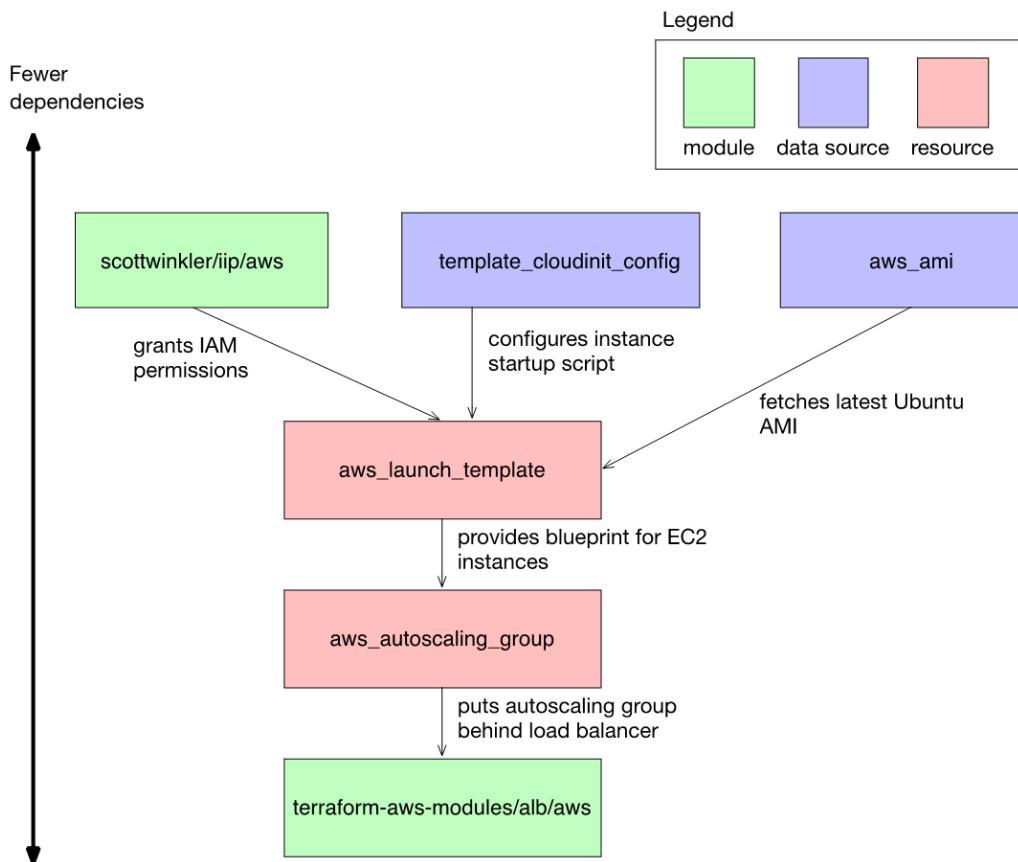


Figure 4.16 initial dependency diagram of managed and unmanaged resources in autoscaling module

I find these sorts of sketches to be useful when planning out the code of a Terraform module. Even after my code is written, I find them to be much more helpful than the diagrams generated by `terraform graph` at visualizing my infrastructure. Whenever I am planning out the code for a Terraform module, I always consider inter-resource dependencies (i.e. what depends on what) because it helps me predict potential race conditions that require an explicit `depends_on`. These sketches are often incomplete or outright wrong, but they do provide a starting point from which to work off of. After we're finished writing the code in this section, we'll compare what my initial dependency diagram looks like vs. what the actual dependency diagram is.

Another thing I like to do is write the Terraform code from top-to-bottom, in a way that matches how my dependency diagram look like; i.e. resource having fewer dependencies are put at the top of the file, and resources having more dependencies are put at the bottom. Therefore, you can reason that resources at the top of the file will be created before the

resources at the bottom of the file, kind of like “normal” procedural code. Technically you do not need to do this, because Terraform will optimize and organize resources for you when it generates an execution plan, but I find it helpful anyways. In particular, it helps me understand what my code is doing and anticipate how it will behave at runtime. Many other Terraform developers in the community will do this without even realizing it, because it’s such a natural way of thinking. Now that we’re finished with the detailed planning, let’s start writing the code.

### 4.7.3 Getting Real with Template Files

Remember how I said that templates are useful for init scripts? Here is a case in point. Listing 4.16 showcases the code for creating a launch template based on some hardcoded input configuration, as well as an IAM instance role and a cloud init configuration.

**Listing 4.16 main.tf**

```
module "iam_instance_profile" { #A
  source  = "scottwinkler/iip/aws"
  actions = ["logs:*", "rds:*"] #B
}

data "template_cloudinit_config" "config" {
  gzip        = true
  base64_encode = true
  part {
    content_type = "text/cloud-config"
    content      = templatefile("${path.module}/cloud_config.yaml", var.db_config) #C
  }
}

data "aws_ami" "ubuntu" {
  most_recent = true
  filter {
    name   = "name"
    values = ["ubuntu/images/hvm-ssd/ubuntu-bionic-18.04-amd64-server-*"]
  }
  owners = ["099720109477"]
}

resource "aws_launch_template" "webserver" {
  name_prefix  = var.namespace
  image_id     = data.aws_ami.ubuntu.id #D
  instance_type = "t2.micro"
  user_data    = data.template_cloudinit_config.config.rendered #D
  key_name     = var.ssh_keypair
  iam_instance_profile {
    name = module.iam_instance_profile.name #D
  }
  vpc_security_group_ids = [var.sg.websvr]
}
```

#A A module I published for creating iam\_instance\_profiles based on a list of permissions

#B rds:\* is too open, you would not want to do this in production

#C Content for the cloud init configuration comes from a template file

```
#D Specifying implicit dependencies between 1) iam_instance_profile, aws_ami, and template_cloudinit_config and 2)
aws_launch_template
```

Notice that the cloud init configuration is templated using the `templatefile` function we saw in chapter 3. This function accepts two arguments, a path to a template file named `cloud_config.yaml`, and a `variables` object referenced from `var.db_config`. We use the special interpolation variable `path.module` to get a reference to the relative filesystem path. The result of this function is the configuration that the web server needs to be able to connect with the database when it starts up. The code for `cloud_config.yaml` is shown in Listing 4.17.

#### **Listing 4.17 cloud\_config.yaml**

```
#cloud-config
write_files:
  - path: /etc/server.conf
    owner: root:root
    permissions: "0644"
    content: |
      {
        "user": "${user}", #A
        "password": "${password}", #A
        "database": "${database}", #A
        "netloc": "${hostname}:${port}" #A
      }
runcmd: #B
  - curl -sL https://api.github.com/repos/scottwinkler/vanilla-webserver-src/releases/latest
    | jq -r ".assets[].browser_download_url" | wget -qi -
  - unzip deployment.zip
  - ./deployment/server
packages:
  - jq
  - wget
  - unzip
```

#A the content of this file will be templated by the db\_config object

#B downloading the web application code and starting the server

This is a pretty normal cloud init file. All it does is install some packages, create a configuration file (`/etc/server.conf`), fetch the application code (`deployment.zip`) and start the server.

#### **4.7.4 Final Touches**

Finally, we're ready to add the code for the launch template, autoscaling group and load balancer to `main.tf`. The code in Listing 4.18 shows how to do this. Don't worry too much about what the values mean, these can all be found in the AWS provider documentation, or the module `README.md`. Most are just default or "safe" values chosen for the purposes of this exercise. Instead, pay attention to how data flows from the other modules we defined into the resources and modules declared here. That's what I mean by "trickling down".

**Listing 4.18 main.tf**

```

module "iam_instance_profile" {
  source  = "scottwinkler/iip/aws"
  actions = ["logs:*", "rds:*"]
}

data "template_cloudinit_config" "config" {
  gzip      = true
  base64_encode = true
  part {
    content_type = "text/cloud-config"
    content      = templatefile("${path.module}/cloud_config.yaml", var.db_config)
  }
}

data "aws_ami" "ubuntu" {
  most_recent = true
  filter {
    name   = "name"
    values = ["ubuntu/images/hvm-ssd/ubuntu-bionic-18.04-amd64-server-*"]
  }
  owners = ["099720109477"]
}

resource "aws_launch_template" "webserver" {
  name_prefix      = var.namespace
  image_id         = data.aws_ami.ubuntu.id
  instance_type    = "t2.micro"
  user_data        = data.template_cloudinit_config.config.rendered
  key_name          = var.ssh_keypair
  iam_instance_profile {
    name = module.iam_instance_profile.name
  }
  vpc_security_group_ids = [var.sg.websvr]
}

resource "aws_autoscaling_group" "webserver" {
  name           = "${var.namespace}-asg" #A
  min_size       = 1
  max_size       = 3
  vpc_zone_identifier = var.vpc.private_subnets
  target_group_arns = module.alb.target_group_arns
  launch_template {
    id = aws_launch_template.webserver.id #B
    version = aws_launch_template.webserver.latest_version #B
  }
}

module "alb" {
  source  = "terraform-aws-modules/alb/aws"
  version = "~> 4.0"
  load_balancer_name      = "${var.namespace}-alb"
  security_groups          = [var.sg.lb] #C
  subnets                  = var.vpc.public_subnets
  vpc_id                   = var.vpc.vpc_id
  logging_enabled          = false
  http_tcp_listeners        = [{ port = 80, protocol = "HTTP" }]
  http_tcp_listeners_count = "1"
}

```

```

target_groups          = [{ name = "websvr", backend_protocol = "HTTP", backend_port =
    8080 }]
target_groups_count    = "1"
}

```

#A Using the namespace variable to prevent resource name collisions  
#B The autoscaling group will always use the latest launch template version  
#C Security group gets set here after traveling from the networking module

**WARNING** Exposing port 80 over HTTP for a publicly facing load balancer is unacceptable security for production level applications. Always use port 443 over HTTPS with an SSL/TLS certificate!

Now that we're done, we can draw a new dependency diagram based on what actually exist  
Our actual dependency diagram actually looks more like figure 4.17.

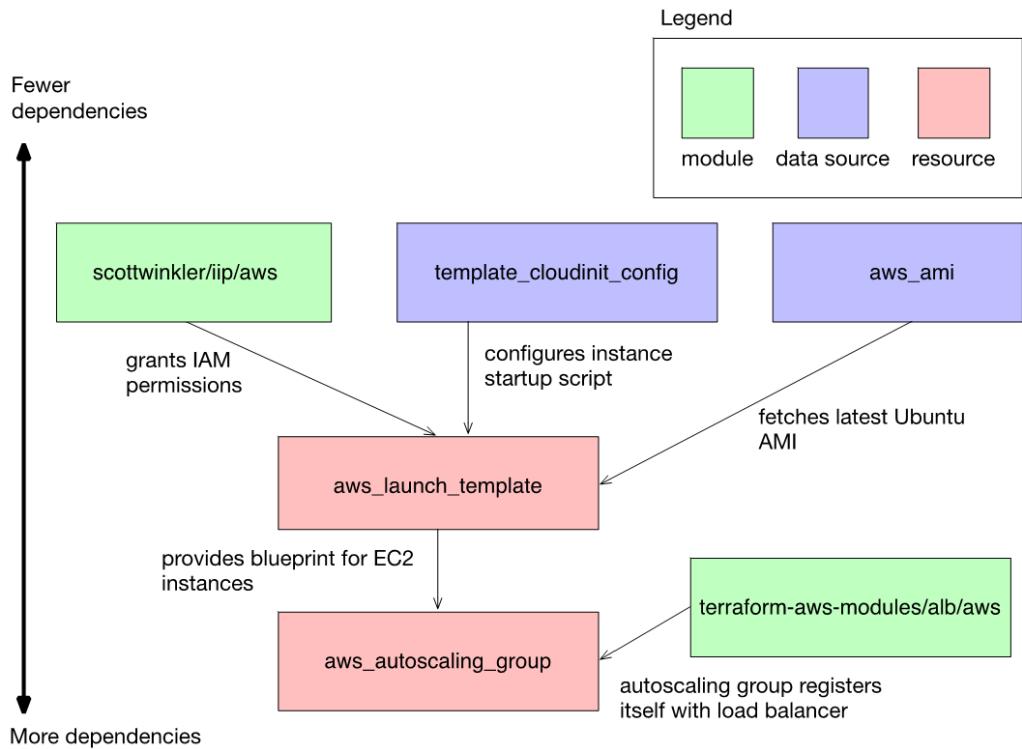


Figure 4.17 revised dependency graph with the autoscaling group depending on the load balancer

The discrepancy between how you plan a module vs. what you end up with is a common occurrence when working with Terraform. Does it really matter whether the autoscaling group registers itself with the load balancer or it's the load balancer doing the registering? I would

say no, it doesn't matter, because it's Terraform's job to manage dependencies, not you. As long as the code does what it is supposed to and it's organized in a way that's easy to understand, you've done a great job.

**TIP** besides organizing code top-to-bottom based from least to most dependencies, you can also organize code by grouping related resources together in the same file with a multi-line comment block header describing the groups purpose.

Lastly, there is an output of the module, `lb_dns_name`, that we need to include. This output is used to make it easier to find the DNS name after deploying and is bubbled up to the output of the root module. Only outputs from the root module show up in the command line after applying. Listing 4.19 has the code for `outputs.tf`.

#### **Listing 4.19 outputs.tf**

```
output "lb_dns_name" {
    value = module.alb.dns_name
}
```

We can make this output available from the root module by adding another output value to passthrough the data.

#### **Listing 4.20 outputs.tf in root module**

```
output "db_password" {
    value = module.database.db_config.password
}

output "lb_dns_name" {
    value = module.autoscaling.lb_dns_name
}
```

## 4.8 Deploying the Webapp

We've created a lot of files, which is not unusual with Terraform, especially when componentizing code into modules. The complete list of files for the scenario is listed below for reference.

```
$ tree
.
├── main.tf
└── modules
    ├── autoscaling
    │   ├── cloud_config.yaml
    │   ├── main.tf
    │   ├── outputs.tf
    │   └── variables.tf
    ├── database
    │   ├── main.tf
    │   ├── outputs.tf
    │   └── variables.tf
    └── networking
```

```

    └── main.tf
    └── outputs.tf
        └── variables.tf
    ├── outputs.tf
    ├── providers.tf
    ├── terraform.tfvars
    └── variables.tf
    └── versions.tf

```

4 directories, 16 files

At this point, we're ready to deploy the webapp to the cloud. Change directory into the root module and run a `terraform init` followed by a `terraform apply -auto-approve`. After waiting ~10-15 minutes for all the resources to finish creating, the tail of your output will look as follows:

```

module.autoscaling.aws_autoscaling_group.webserver: Still creating... [10s elapsed]
module.autoscaling.aws_autoscaling_group.webserver: Still creating... [20s elapsed]
module.autoscaling.aws_autoscaling_group.webserver: Still creating... [30s elapsed]
module.autoscaling.aws_autoscaling_group.webserver: Still creating... [40s elapsed]
module.autoscaling.aws_autoscaling_group.webserver: Creation complete after 48s [id=my-cool-project-asg]

```

Apply complete! Resources: 40 added, 0 changed, 0 destroyed.

Outputs:

```

db_password = yRTKf1ZHJwUgA-1F #A
lb_dns_name = my-cool-project-alb-956291277.us-west-2.elb.amazonaws.com #A

```

#A your db\_password and lb\_dns\_name will be different from mine

Now copy the value of `lb_dns_name` into your web browser of choice to navigate to the website. If you get a 502 bad gateway error, wait a few more seconds before trying again, as it hasn't finished initializing yet. Figure 4.16 depicts what the final website looks like. You can click the "+" button to add pictures of your cats or other animals to the database, and the animals you add will be viewable by anyone who visits the website.



Figure 4.18 deployed webapp with no pets added yet

When you're done, don't forget to take down the stack to avoid paying for infrastructure you don't need. Do this with a `terraform destroy -auto-approve`. The tail of your output will look as follows (again it will take ~10-15 minutes):

```
module.networking.module.vpc.aws_internet_gateway.this[0]: Destruction complete after 11s
module.networking.module.vpc.aws_vpc.this[0]: Destroying... [id=vpc-0f0ef0db8519a902b]
module.networking.module.vpc.aws_vpc.this[0]: Destruction complete after 1s

Destroy complete! Resources: 40 destroyed.
```

## 4.9 Overview

In this chapter we designed and deployed the Terraform for a push button style web application. We used a well-established, multi-tiered architecture design with some slight alterations tailoring it for the cloud, such as taking advantage of cloud native services like autoscaling groups, elastic load balancing, and a managed MySQL database. The configuration code in this chapter can also be modified to deploy any sort of web application, not just the one specified in `cloud_config.yaml`. It's a robust and scalable design that will work well for porting a wide variety of legacy applications directly to the cloud.

Componentizing your software systems using modules is the key takeaway from this chapter. This is an established best practice for developing complex deployments in Terraform. The main advantage of separating code into smaller components is that it is easier to statically analyze and reason about your code when you can treat them as logical black boxes, but it does force you to create a lot of small files with not much code in them, and some boilerplate for passing data between modules. In the next chapter we'll look at an alternative way to organize code, using a single flat root module, rather than many nested submodules.

A general diagram depicting the software componentization pattern is shown in figure 4.17.

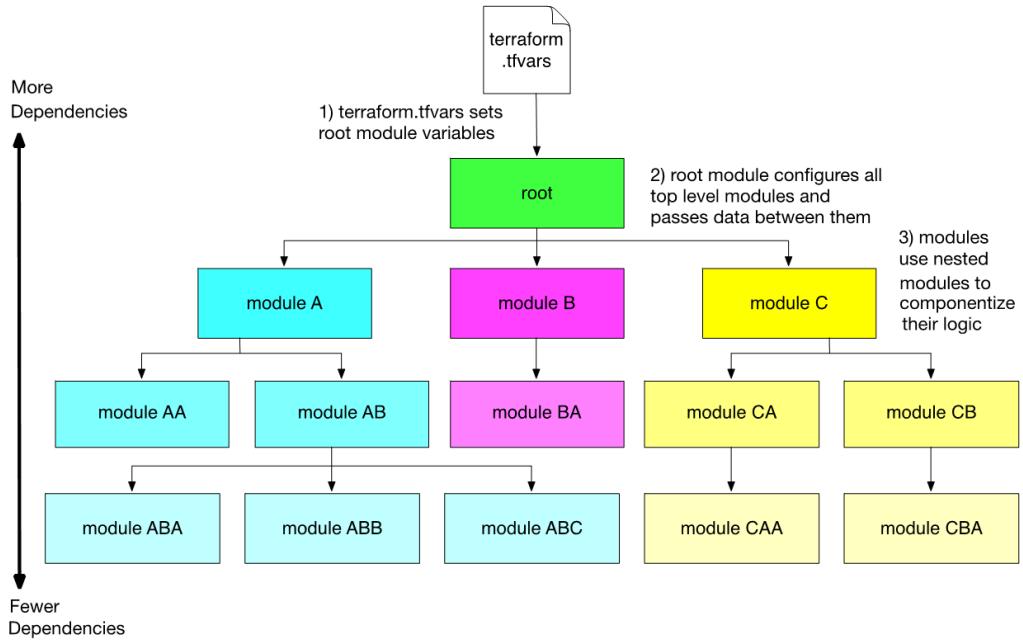


Figure 4.19 generalized software componentization pattern for organizing code using modules

## 4.10 Summary

- Modules can be either used for code organization or code reuse. The source may be either a local directory (e.g. `/modules/<module_name>`) or a remote location such as the Terraform module registry.
- As directed by HashiCorp, it is best practice that each module has at least the following three files: `main.tf`, `variables.tf` and `outputs.tf`.
- The root module serves as the entry point for your project, thus necessitating three additional files: a variables definition file (`terraform.tfvars`), `versions.tf` and `providers.tf`. The variables definition file is used to set common configuration for top-level modules.
- Terraform projects should be componentized into logical groupings of systems based on the principle of least privilege. Pass in only the minimum amount of data that a module will need to do its job, no more, no less.
- All data passed between modules will either be bubbled up or trickled down. Optimizing the way data flows between modules by reducing the distance between where data is produced and where it is consumed will help you to organize your code better.

# 5

## *Serverless Made Easy*

### **This chapter covers:**

- Deploying a serverless application with Azure
- Problem solving techniques and strategies for Terraform projects
- Adapting modules to fetch build artifacts at runtime
- Overcoming deficiencies and bugs in the Azure provider
- Case studies for combining Azure Resource Manager (ARM) with Terraform

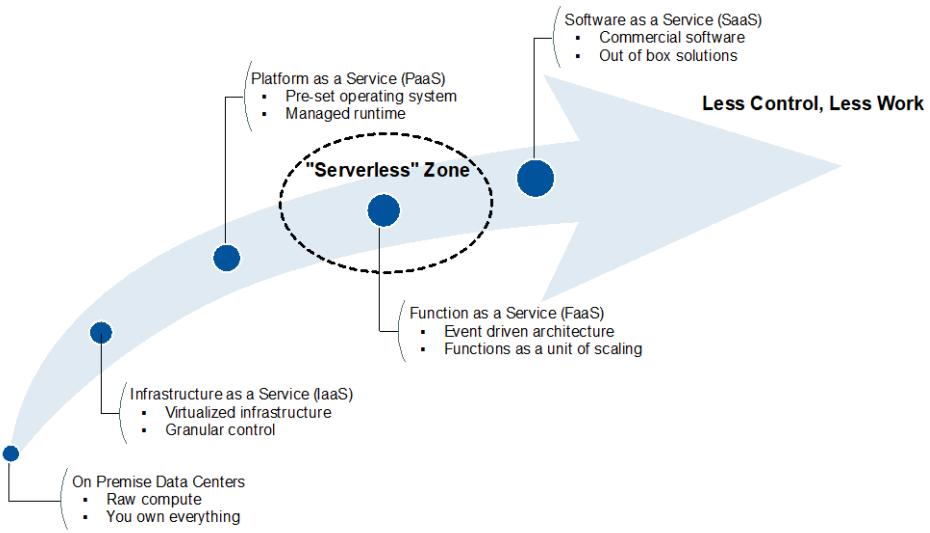
*Serverless* is one of the biggest marketing gimmicks of all time. It seems like everything is marketed as “serverless” these days, despite nobody being able to agree on what the word even means. One thing for sure is that serverless does **not** refer to the elimination of servers; in fact it usually means the opposite. Serverless implementations typically involve dynamic, scalable, and highly distributed systems, so there’s often way more servers involved than in conventional system designs. Another thing to be aware of is that “serverless” is an umbrella term for technologies sharing two key characteristics:

1. pay-as-you-go billing
2. minimal operational overhead

Pay-as-you-go billing is all about paying for the actual amount of resources consumed, rather than pre-purchased units of capacity (i.e. pay for what you use, not what you don’t use). Yes, you could save money by buying reserved units of capacity, but it’s easy to overestimate how much you’ll really need and end up paying way too much. Like when you go to the movies and buy a huge bag of popcorn because its only 50 cents more, but you actually would have been fine with a small.

Minimal operational overhead means that the cloud provider takes on most or all of the responsibility of scaling, maintaining and managing the service. You only have to consume the service, and not worry about how to run it. The burden of maintaining server-side software and applying patches and so on moves from you to the cloud provider. Less effort on your part, for sure, but also less control. Some people argue that the term “serverless” is accurate, because the fact that servers are involved at all is purely an implementation detail from a user’s perspective. While I accept this is a valid argument, I do find it misleading. Like how Tic-Tacs can be labeled as “sugar free” since they contain less than 0.5 grams of sugar per serving, even though they are nearly 100% pure sugar. If servers are being used anywhere, then it is my opinion that a service cannot truly be “serverless”.

Despite the confusing vernacular, it’s impossible to deny the rise in popularity of serverless technologies in recent years. One question many people have is how serverless compares to other infrastructure technologies. The way I like to think of it is a tradeoff between the level of control you want, and the amount of work you’re willing to invest. If there was a spectrum of control vs work, on-premise data centers would represent the point of most control, while also requiring the greatest investment of work. Meanwhile, Software as a Service (SaaS) is the opposite: it grants the least amount of control but demands little work. Between these two extremes, serverless would be much closer to SaaS than on-prem. Figure 5.1 illustrates where serverless falls on the spectrum compared to other popular technologies. Note that because serverless actually refers to an umbrella of different technologies, I’ve drawn it as a zone rather than a point, because some serverless technologies are more akin to SaaS products while some are more like PaaS.



More Control, More Work

Figure 5.1 “Serverless” is a region of space existing somewhere between PaaS and SaaS

In this chapter we’re going to be deploying an Azure Functions website with Terraform. *Azure Functions* (a.k.a. Azure Functions App) is a FaaS technology, similar to AWS Lambda or Google Cloud Functions which allow you to write stateless and infinitely scalable functions for the business logic of your application. The specific scenario we’ll be looking at is a serverless multi-tier web application. It’s going to have a NoSQL database, a RESTful API, and static content hosted entirely on serverless infrastructure. Does this sound familiar? It should, because it’s almost exactly the same as what we deployed in the previous chapter. This is a fun scenario that highlights the contrast between the bulky infrastructure required for a monolithic deployment and its lightweight serverless counterpart. Serverless has come a long way in recent years, and now it’s capable of doing almost anything that used to require a dedicated virtual machine.

### Functions are Atomic

Functions are atomic units of logic, which means they cannot be further broken down without losing meaning. They have a number of distinct advantages, namely, being easy to reason about, easy to unit test, and easy to scale (especially if they are stateless). But nothing comes for free. It’s important to recognize that by breaking your application into functions, you’re accepting a tradeoff between decreased code complexity and increased wiring requirements between components. In other words, highly decoupled (i.e. serverless) software architectures suffer from complexity issues that highly coupled (i.e. monolithic) software architectures do not, and vice-versa.

To understand why this is the case, consider the differences that exist between monolithic applications and microservices. Monolithic applications run on VM’s or bare metal hardware and do not require much – or really any – wiring between components. There’s little need for fancy event driven architectures when everything can talk directly to

everything else. Of course, the disadvantage of monolithic applications is that they are much harder to reason about (among other adverse qualities). Microservices have more wiring involved but are popular mainly because they are faster to iterate on, and deliver new features with, compared to equivalent monolithic applications.

Functions go a step further than microservices by dividing *all* of the logic for an application into a series of independent functions (see figure 5.2). Now, you have virtually no coupling between components, and it becomes very easy to reason about what your code is doing when you only ever have to consider the inputs and outputs of each function to understand how they work. As previously mentioned, functions have the drawback of requiring substantially more wiring between components. Fortunately, however, the cloud provider does most of this heavy lifting for you. Serverless applications are a great new technology, and are definitely worth looking into.

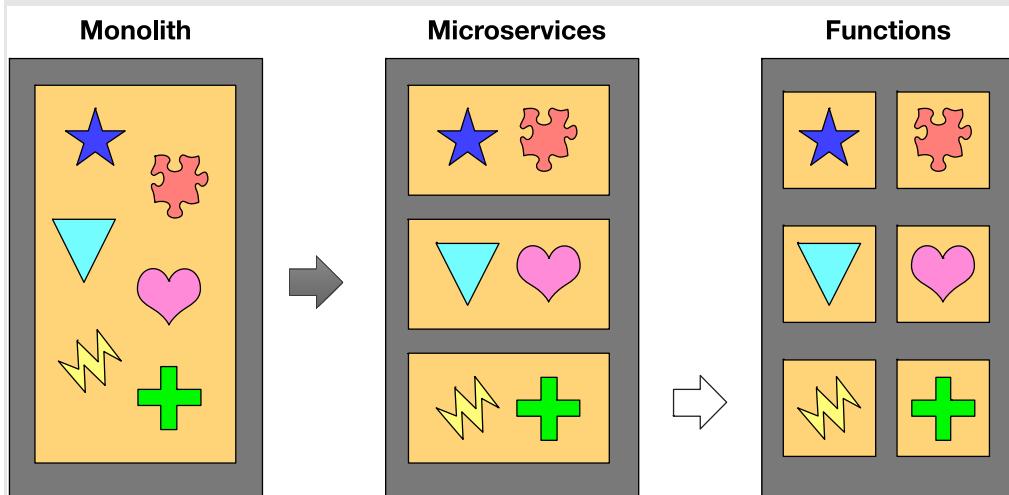
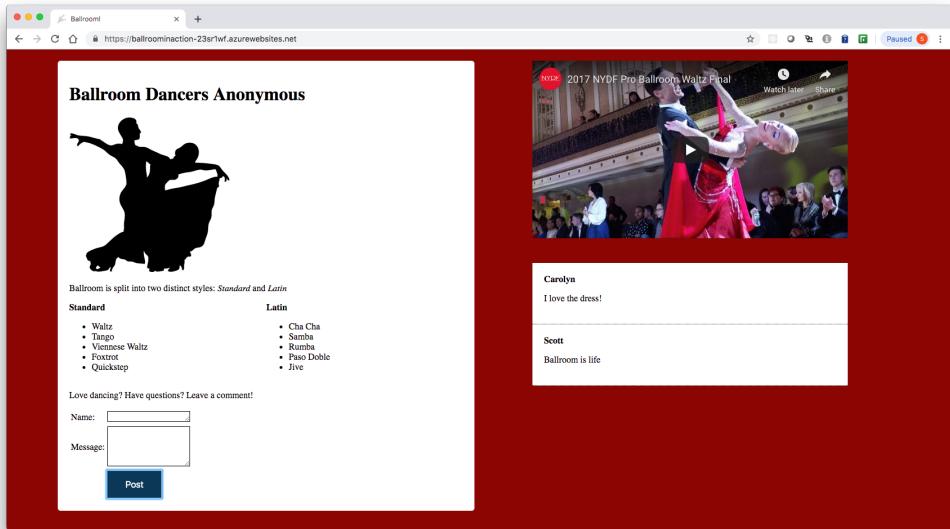


Figure 5.2 Functions are the natural result of breaking up the monolith

## 5.1 The “Two Penny Website”

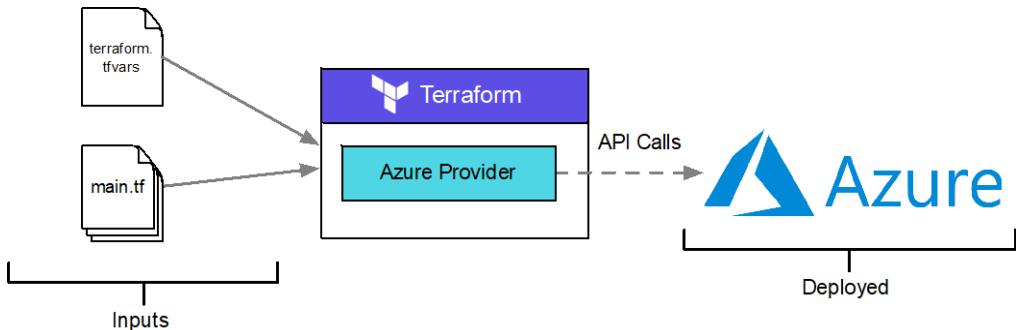
This scenario is something I like to call “the two-penny website”, because that’s how much I estimate it will cost to run on a monthly basis. If you can scrounge some coins from between the seat cushions, you’re good for at least a year or two of web hosting. For most low traffic websites, the true cost will likely be even less, perhaps a mere fraction of one penny. If this still sounds too pricey for you, Azure has generous free-tier and student level accounts to help you to get started (AWS and GCP have their equivalents).

The website we are going to be deploying is a ballroom dancing forum, called “Ballroom Dancers Anonymous”. Unauthenticated users will be able to leave comments which are stored in a database and are viewable by other users on the site. The design is fairly simple, but the beauty is that what I am giving you can be generalized to work with a wide variety of web applications. A sneak peak of the final product is shown in figure 5.3.



**Figure 5.3** Ballroom Dancers Anonymous Website

We will be using Azure to deploy the serverless website, but it shouldn't feel any different than deploying to AWS. A basic deployment strategy is shown in figure 5.4.



**Figure 5.4** Deploying to Azure is no different than deploying to AWS

## 5.2 Architecture and Planning

Although this website cost only pennies to run, by no means is it a toy. Because it's deployed on Azure Functions, it's able to rapidly scale out to handle tremendous spikes in traffic and do so with low latency. It also uses HTTPS (which is something the previous chapter did not use),

a NoSQL database, and serves both static content (HTML/CSS/JS) as well as a fully-fledged RESTful API. An architecture diagram is shown in figure 5.5.

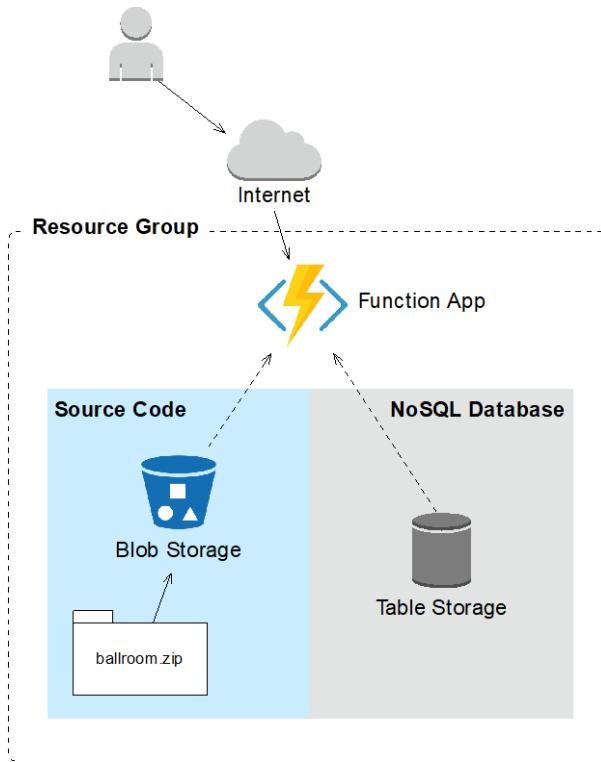


Figure 5.5 Azure Function App listens for HTTP requests coming from the internet. When a request is made, it starts a just-in-time webserver from source code located in a storage container. All stateful data is stored in a NoSQL database using a service called Azure Table Storage.

### 5.2.1 Sorting by Group, then by Size

Because the code we'll be writing is relatively short and cohesive, its best to put it all in a single `main.tf` file instead of using nested modules.

**TIP** As a rule of thumb, I would suggest having no more than a few hundred lines of code per Terraform configuration file. Any more and it becomes difficult to build a mental map of how the code actually works. Of course, the exact number is for you to decide.

The big question then becomes: how should the code be organized so that it's both easy to read and easy to understand? We could just throw resources into `main.tf` and call it done, but that could be a source of tech debt in the future. Future maintainers of the code may not understand

why the sequence of resources being created is the way it is. On the other hand, spending time to organize code can take significantly more time up front. It all depends on how many times you think the code will be written vs. read. Generally, code will be read many times and only written once. So, writing, structuring, and naming your code really matters.

As discussed in chapter 4, organizing code based on the number of dependencies is always a sound approach. This means resources having fewer number of dependencies are located toward the top of the file, and vice versa. However, strictly following this rule leaves a lot of room for ambiguity, especially when two resources have the same number of dependencies, but don't feel like they *belong together*.

### **Grouping Resources that Belong Together**

By "belonging together", I mean the intuitive sense that things are either related or they are not. Sorting resources purely by number of dependencies is not always the best idea. For example, if you had a bag of multicolored marbles, sorting from smallest to largest marble size might be a good starting point, but it wouldn't help you find marbles of a particular color. You could first group marbles by color, then sort by size, and finally organize the groups so that the overall trend follows increasing marble size (see figure 5.6)

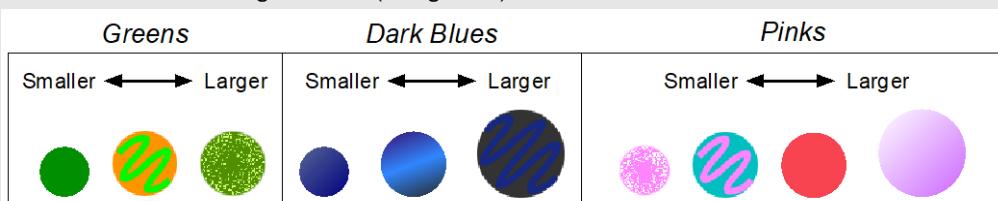
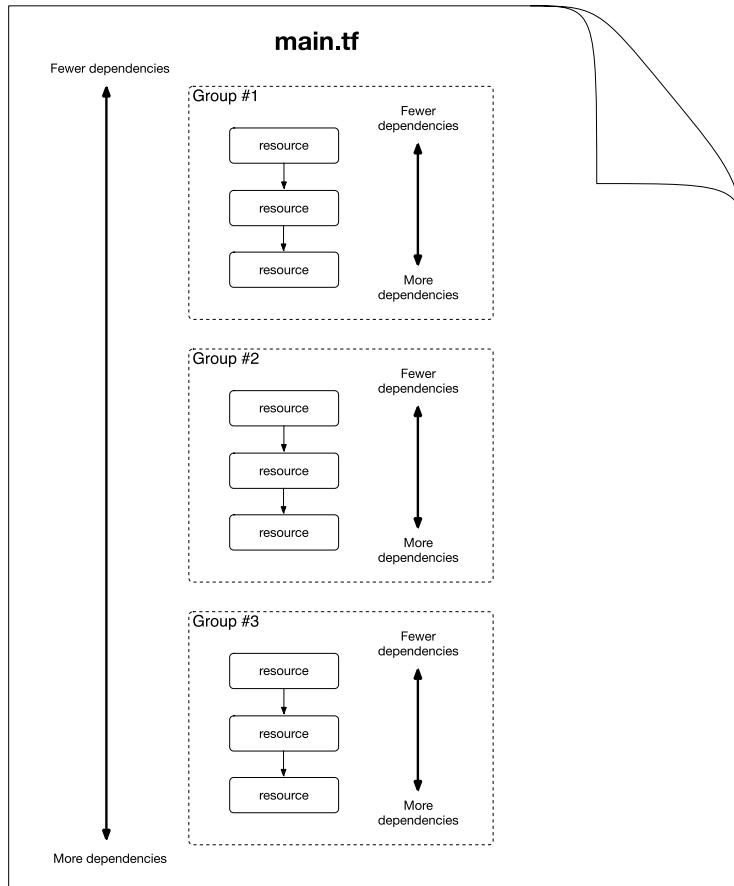


Figure 5.6 Sorting marbles with respect to size and color. Generally, size increases as you go from left to right, but there are exceptions.

The real world is a messy place and doesn't usually lend itself well to categorization. To continue with the analogy of marbles, what do you with a marble that doesn't fit into one of the color groups you have defined? Does an orange marble with a green stripe qualify as "green"? It might, it all depends on how you define green. Creating logical groups for resources is highly subjective, and what makes sense to you may not make sense to someone else.

The idea of organizing by some characteristic other than the number of resource dependencies (henceforth called "size") is a common strategy when writing clean Terraform code. The idea is to first group related resources together, then sort each of these groups by size, and finally organize the groups in such a way that the overall trend is size increasing (see figure 5.7). This makes your code both easy to read and easy to understand.



**Figure 5.7 Configuration files should be sorted first by group, then by size. Overall trend is size increasing**

In the same way that it's quicker to search for a word in a dictionary than a word search puzzle, it's faster to find what you're looking for when your code is organized in a sensible manner (such as the pattern shown above). For this particular project, I've divided it into four groups, each serving a particular purpose in the overall application deployment. These groups are:

1. **Resource Group** – “resource group” is not the name of this group, but the name of an Azure resource that creates a project container. Resource group, and any other base level resources, are part of a first deployment group that will be at the top of the `main.tf` file. These resources are required by many downstream resources but have no resource dependencies of their own
2. **Storage container** – the storage container is part of a second group that is used to store the build artifact (or source code) for Azure Functions App. It also serves a dual function as the NoSQL database for the application.

3. **Storage blob** – the storage blob is the third group that uploads the build artifact that Azure Functions App needs to run. If you are familiar with AWS, this is analogous to an object in an S3 bucket.
4. **Azure Functions App** – anything related to deploying and configuring the Azure Functions App is considered part of this group.

The overall architecture is be pictured in figure 5.8.

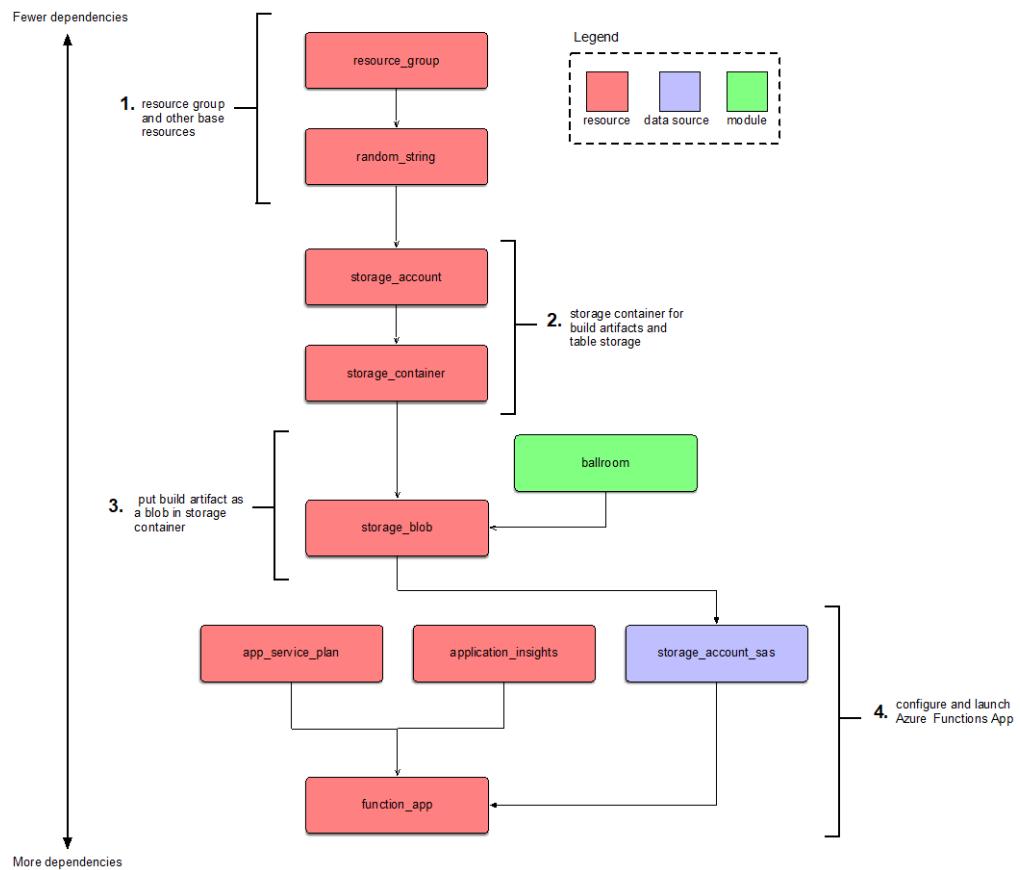


Figure 5.8 There are four main groups in the project, each serving a distinct purpose

The last part of planning we need to do is consider the overall inputs and outputs of the root module. From a black box perspective, there are three inputs and one output. Two of the inputs are used for configuring the provider/base resources (`client_secret` and `region`) and the third is for affording a consistent naming scheme (`namespace`). The sole output value we'll have is

`website_url`, which will be a live link to the final deployed website. This can all be visualized in figure 5.9.

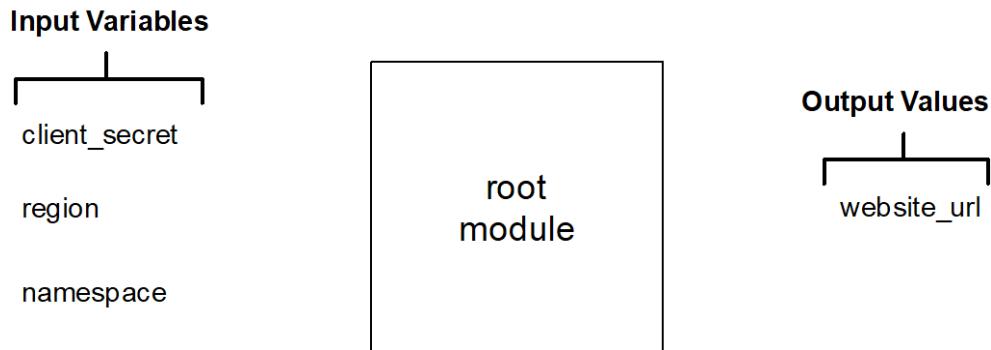


Figure 5.9 Overall input variables and output values of the root module

### 5.2.2 Problem Solving

When planning out this book, I didn't want it to be a bunch of copy-paste snippets because I felt that would be of limited value at best. Terraform relies heavily on cloud services, so as new services are released, a good solution today may not be a good solution tomorrow. Because technology changes so fast, my aim instead is **not** to address every potential use case of Terraform, but to give you a set of tools and techniques that can be applied to solve any kind of problem with Terraform. Here is a list of steps that I suggest taking when tackling a new problem with Terraform:

1. Define the problem and goals
2. Research potential solutions, while keeping an open mind
3. Select key technologies and tools to leverage
4. Build a prototype to determine if further investment is warranted
5. Develop final product

For this particular scenario, my goal was to deploy a serverless website on Azure, but I hadn't used Azure much before (I'm more an AWS guy) so I had to do a lot of research to figure out what Azure services were available to do this. I selected Azure Functions App and Azure Storage because they both seemed to fit my needs. Next, I built a quick and dirty prototype to prove the concept was technically feasible, before finally polishing it up until it became what you see here.

The code in this book is the result of a lot of trial and error. If you want to solve useful problems with Terraform, you also have to be willing to try new things and make mistakes.

**TIP** problem solving is an art, and the only way to get better is with practice

## 5.3 Writing the Code

Recall that there are four groups we need to create:

1. Resource Group
2. Storage Container
3. Storage Blob
4. Azure Functions App

Before jumping into the code, we first need to authenticate to Microsoft Azure and set required input variables.

### Authenticating to Azure

The Azure provider supports four different methods for authenticating to Azure<sup>1</sup>. These are:

Authenticating to Azure using the Azure CLI

Authenticating to Azure using Managed Service Identity

Authenticating to Azure using a Service Principal and a Client Certificate

Authenticating to Azure using a Service Principal and a Client Secret

For this particular scenario, I have chosen method #4, as I found this was the easiest one for me to set up. Although it is also quite easy to use the Azure CLI to authenticate, I would not recommend this method as it is somewhat slow and doesn't work well when running Terraform non-interactively.

When you've obtained credentials to Azure, create a new workspace containing three files: `variables.tf`, `terraform.tfvars`, `providers.tf`. Then insert the contents of Listing 5.1 into `variables.tf`.

### Listing 5.1 variables.tf

```
variable "client_secret" {
  type = string
}

variable "region" {
  type  = string
  default = "westus2"
}
```

<sup>1</sup> <https://terraform.io/docs/providers/azurerm/index.html#authenticating-to-azure>

```
variable "namespace" {
  type  = string
  default = "ballroominaction"
}
```

Next, we will set the variables; listing 5.2 shows the contents of `terraform.tfvars`. Technically, we don't really need to set `region` or `namespace` since the defaults are fine, but it's always a good idea to be thorough.

### **Listing 5.2 terraform.tfvars**

```
client_secret = "00000000-0000-0000-0000-000000000000" #A
region      = "westus2"
namespace   = "ballroominaction"
```

#A you will need to insert your own credentials here

**NOTE** It's fine to keep your secrets in `terraform.tfvars` as long as you don't check the file into version control (by adding this file to `.gitignore` or equivalent). Of course, this won't work for multitenant deployments, but we'll cover this more in the next chapter.

The configured provider for Azure is shown below. You'll need to insert your own `subscription_id`, `client_id`, and `tenant_id`. The Azure documentation<sup>2</sup> can help with this. None of these values are secret, and we aren't going to be sharing this as a module through the module registry, so it's okay to hardcode the values here.

### **Listing 5.3 providers.tf**

```
provider "azurerm" {
  subscription_id = "00000000-0000-0000-000000000000"
  client_id      = "00000000-0000-0000-000000000000"
  client_secret  = var.client_secret
  tenant_id      = "00000000-0000-0000-000000000000"
}
```

#### **5.3.1 Resource Group**

Now we're ready to write the code for the first of the four groups. As is probably obvious from the name of this section, the resource group is one of the resources we'll be creating, but we'll also use this opportunity to provision all other base level resources (spoiler alert: there's only one).

<sup>2</sup> [https://www.terraform.io/docs/providers/azurerm/auth/service\\_principal\\_client\\_secret.html#creating-a-service-principal-using-the-azure-cli](https://www.terraform.io/docs/providers/azurerm/auth/service_principal_client_secret.html#creating-a-service-principal-using-the-azure-cli)

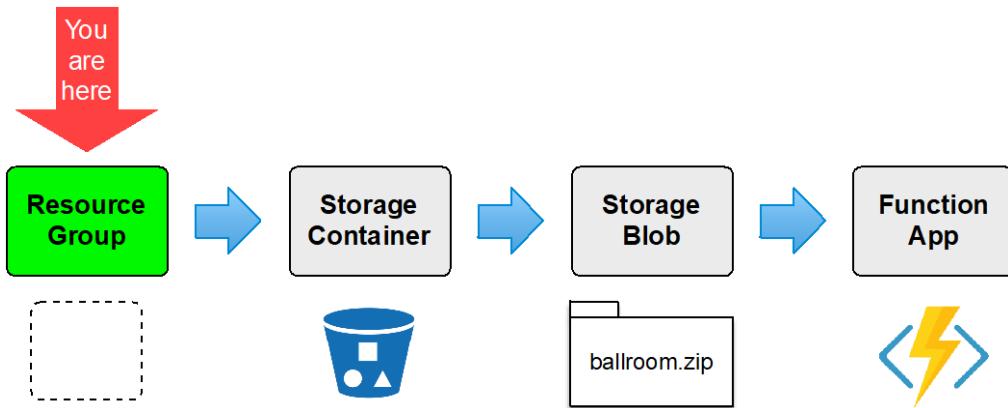


Figure 5.10 Development roadmap – Step 1 of 4

Before we continue, I want to clarify what a resource group is in case you are not familiar with them. In Azure, all resources must be deployed into a resource group, which is essentially a type of container that stores references to resources. Resource groups are convenient, because if a resource group is deleted, all of its children are deleted as well. This synergizes well with Terraform, because you can provision a resource group for each deployment and use these resource groups to create logical environments. As an added benefit, if you happen to lose your terraform state file, you can simply delete the resource group to clean up all the orphaned resources. Resource groups are not unique to Azure (both AWS and Google Cloud have their equivalents<sup>3</sup>), but Azure does require their use. The code for creating a resource group is shown in Listing 5.4.

#### **Listing 5.4 main.tf**

```
resource "azurerm_resource_group" "default" {
  name    = local.namespace
  location = var.region
}
```

Besides the resource group, we'll also want to use the random provider to ensure sufficient randomness beyond what is supplied by the namespace variable. Some resources in Azure must be unique not only in your account, but globally (i.e. across all Azure accounts). The code in Listing 5.5 shows how to accomplish this by joining the namespace variable with the result of

<sup>3</sup> AWS - <https://docs.aws.amazon.com/ARG/latest/userguide/welcome.html>  
GCP - <https://cloud.google.com/storage/docs/projects>

`random_string` and selecting a substring of that of the first 24 characters (effectively adding a right-pad). Add this code right before the `azurerm_resource_group` resource to make the dependency relationship more clear for readers.

#### Listing 5.5 main.tf

```
resource "random_string" "rand" {
  length = 24
  special = false
  upper   = false
}

locals {
  namespace = substr(join("-", [var.namespace, random_string.rand.result]), 0, 24) #A
}
```

#A adding a right-pad to the namespace variable and storing the result in a local value

### 5.3.2 Storage Container

We'll now be using Azure Storage Container for storing the application source code and the documents in the NoSQL database. The NoSQL database is technically a separate service, known as Azure Table Storage, but it's really just a NoSQL wrapper around ordinary key-value access.

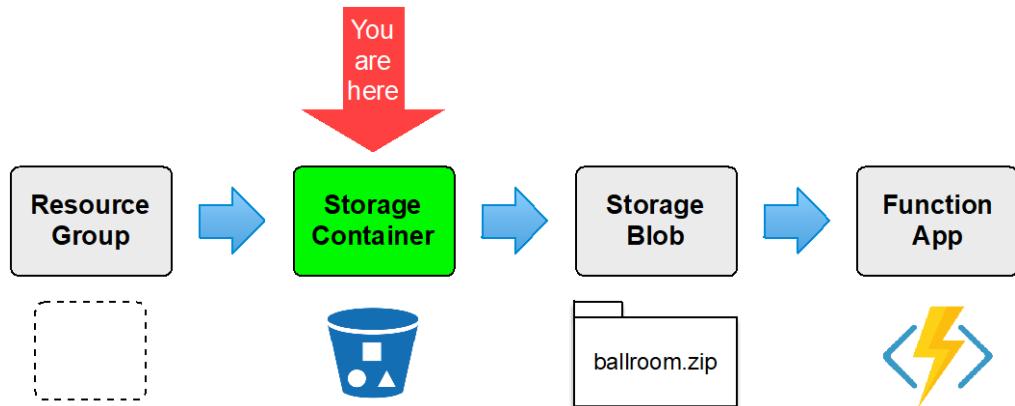


Figure 5.11 Development roadmap – Step 2 of 4

Provisioning a container in Azure is a two-step process. First you need to create a storage account, which provides some metadata about where the data will be stored and how much redundancy/data replication you'd like; I just stick with standard values because it's a good balance between cost and durability. Second, you need to create the container itself. Below is the code for both of these.

**Listing 5.6 main.tf**

```
resource "azurerm_storage_account" "storage_account" {
  name          = random_string.rand.result
  resource_group_name = azurerm_resource_group.default.name
  location       = azurerm_resource_group.default.location
  account_tier    = "Standard"
  account_replication_type = "LRS"
}

resource "azurerm_storage_container" "storage_container" {
  name          = "serverless"
  storage_account_name = azurerm_storage_account.storage_account.name
  container_access_type = "private"
}
```

**NOTE** This would be the place to add an additional container for static website hosting in Azure Storage, if you wanted to do that. For this project, it won't be necessary because Azure Functions will serve up all the static content instead

**Why not use static website hosting in Azure Storage?**

While it is possible – and even recommended – to use Azure Storage as a CDN for hosting your static web content, unfortunately it isn't possible for the Azure provider to do this at this time. Some people have skirted around the issue by using local-exec resource provisioners, but this isn't best practice. Chapter 7 will cover resource provisioners in-depth.

### 5.3.3 Storage Blob

One of the things I like best about Azure Function App is that it gives you so many different options when it comes to how you want to deploy your source code. For example, you could:

1. Use the Azure Functions CLI tool
2. Manually edit the code using the UI
3. Use an extension for VS Code
4. Run from a zip package referenced with a publicly accessible URL

For this scenario, we'll be doing method #4 (running from a zip package referenced with a publicly accessible URL), because it allows us to deploy the project with a single `terraform apply` command.

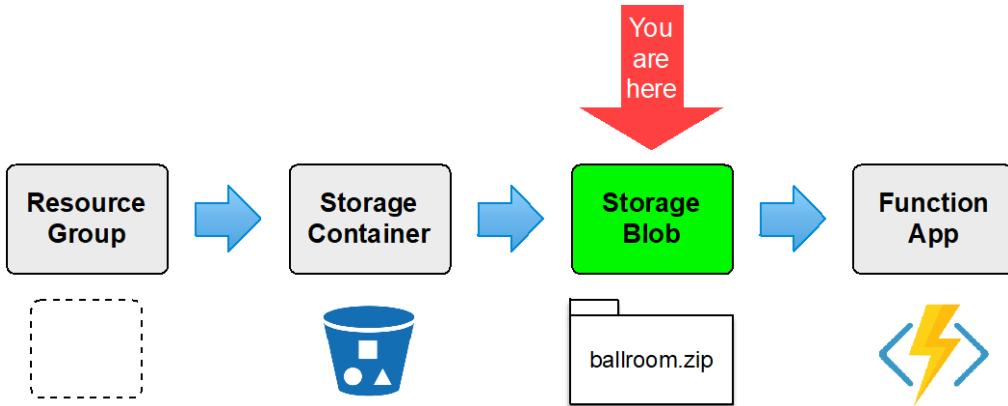


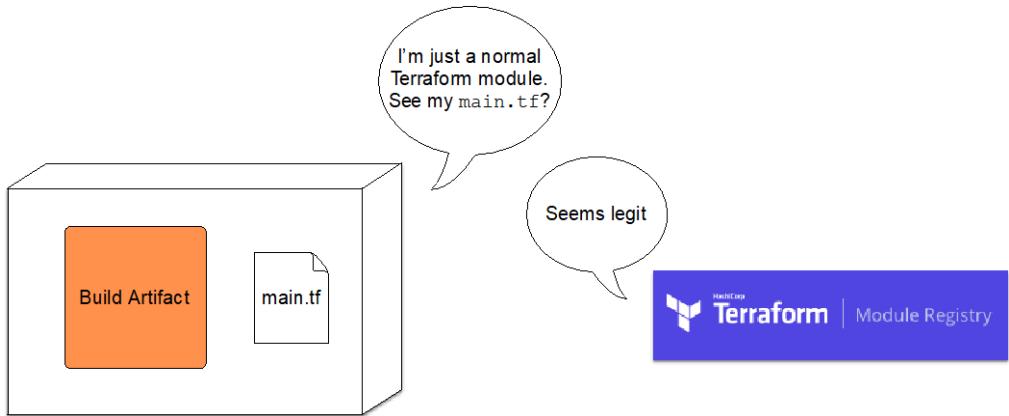
Figure 5.12 Development roadmap – Step 3 of 4

You may be asking, “where does the source code zip file come from?”. Normally the source code would be something you would already have locally on your machine, or as a build artifact stored remotely. Since I wanted this to work with no additional steps, I’ve packaged the code into a Terraform module instead.

As you’ve already learned, remote modules can be fetched from the Terraform Module Registry on `terraform init` or `terraform get`. However, it’s not just Terraform configuration files that are downloaded when you do this, *everything* stored in the module is downloaded together. The entire application source code for this project is stored as a shim module that I’ve registered in the Terraform Module Registry (which we’ll talk about next chapter). Figure 5.13 illustrates how this was done.

**TIP** In a production environment you would either store a copy of the build artifact with your configuration code, or you would fetch it dynamically before running an `apply`, for example as part of a CI/CD pipeline

**WARNING** modules can execute malicious code on your local machine by taking advantage of local-exec provisioners. They can also do harmful things in your cloud account like starting instances running mining operations. You should always skim through the source code of a untrusted module before deploying it



**Figure 5.13 Registering a shim module with the Terraform Module Registry**

The shim module is simply a mechanism for downloading the build artifact onto your local machine, so it can then be uploaded to blob storage. Please add the following code from Listing 5.7 into `main.tf` to do this.

#### **Listing 5.7 main.tf**

```
module "ballroom" {
  source = "scottwinkler/ballroom/azure"
}

resource "azurerm_storage_blob" "storage_blob" {
  name        = "server.zip"
  storage_account_name = azurerm_storage_account.storage_account.name
  storage_container_name = azurerm_storage_container.storage_container.name
  type        = "block"
  source      = module.ballroom.output_path
}
```

#### **5.3.4 Function App**

I wish I could say it was all smooth sailing from here on out, but sadly that is not the case. The Function App needs to be able to download the application source code from the private storage container, which requires a URL presigned by a Shared Access Signature (SAS Token).

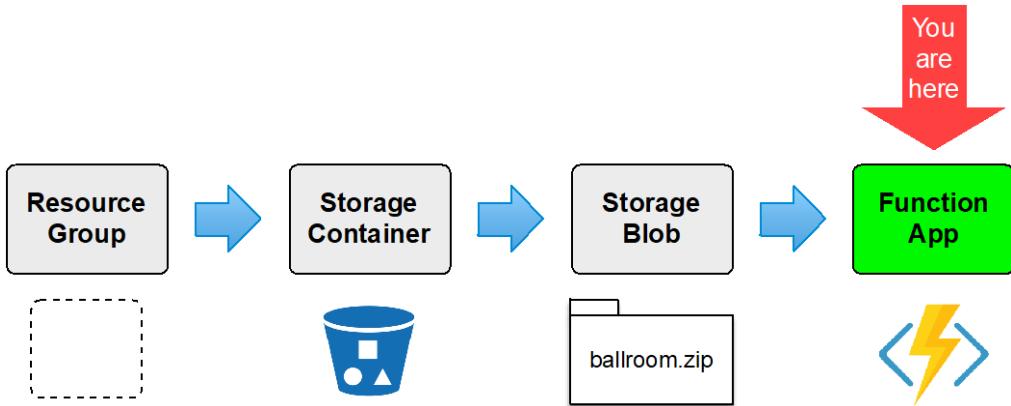


Figure 5.14 Development roadmap – Step 4 of 4

Lucky for us, there is a data source for being able to produce the SAS token with Terraform. The code from Listing 5.8 creates a token that allows the invoker to read from an object in the container with an expiry date set in 2048 (Azure Functions continuously uses this token to download the storage blob, so the expiry must be set into the far future)

#### **Listing 5.8 main.tf**

```
data "azurerm_storage_account_sas" "storage_sas" {
  connection_string = azurerm_storage_account.storage_account.primary_connection_string

  resource_types {
    service = false
    container = false
    object = true
  }

  services {
    blob = true
    queue = false
    table = false
    file = false
  }

  start = "2016-06-19T00:00:00Z"
  expiry = "2048-06-19T00:00:00Z" #A

  permissions {
    read = true
    write = false
    delete = false
    list = false
    add = false
    create = false
    update = false
  }
}
```

```

    process = false
}
}

```

#A this should be set for the far future

Now that we have the SAS token, we have to generate the presigned URL. It would be nice if there was a data source to do this, but sadly there is not. It's kind of a long calculation, so I took the liberty of setting it to a local value for readability purposes. Add this into `main.tf`:

**NOTE** There is currently a bug with the way `azurerm_storage_account_sas` data source generates SAS Tokens. Specifically, it does not properly URL encode the signature. I have solved this by using the `replace()` function to find and replace bad characters.

### Listing 5.9 main.tf

```

locals {
  package_url =
    replace("https://${azurerm_storage_account.storage_account.name}.blob.core.windows.net/${azurerm_storage_c
  ontainer.storage_container.name}/${azurerm_storage_blob.storage_blob.name}${data.azurerm_storage_account_s
  as.storage_sas.sas}", "%3d", "=") #A
}

```

#A this ugliness is how you generate a presigned URL for a blob in Azure Storage

Finally, add the code for creating an `azurerm_application_insights` resource (used for instrumentation and logging), and the `azurerm_function_app` resource.

### Listing 5.10 main.tf

```

resource "azurerm_app_service_plan" "plan" {
  name      = local.namespace
  location   = azurerm_resource_group.default.location
  resource_group_name = azurerm_resource_group.default.name
  kind       = "functionapp"
  sku {
    tier = "Dynamic"
    size = "Y1"
  }
}

resource "azurerm_application_insights" "application_insights" {
  name      = local.namespace
  location   = azurerm_resource_group.default.location
  resource_group_name = azurerm_resource_group.default.name
  application_type = "Web"
}

resource "azurerm_function_app" "function" {
  name      = local.namespace
  location   = azurerm_resource_group.default.location
  resource_group_name = azurerm_resource_group.default.name
  app_service_plan_id = azurerm_app_service_plan.plan.id
}

```

```

https_only      = true
storage_connection_string = azurerm_storage_account.storage_account.primary_connection_string
version         = "~2"
app_settings = {
  FUNCTIONS_WORKER_RUNTIME    = "node"
  WEBSITE_RUN_FROM_PACKAGE   = local.package_url #A
  WEBSITE_NODE_DEFAULT_VERSION = "10.14.1"
  APPINSIGHTS_INSTRUMENTATIONKEY = azurerm_application_insights.application_insights.instrumentation_key
  TABLES_CONNECTION_STRING    = data.azurerm_storage_account_sas.storage_sas.connection_string #B
  AzureWebJobsDisableHomepage = true
}
}

```

#A This points to the build artifact

#B This allows the app to connect to the database

### 5.3.5 Final Touches

We're in the home stretch! All we need to do now is version lock the providers and set the output variable so that we'll have an easy link to the deployed website. Create a new file called `versions.tf` and insert the following code from Listing 5.10:

#### **Listing 5.11 versions.tf**

```

terraform {
  required_version = "~> 0.12"
  required_providers {
    archive = "~> 1.2"
    azurerm = "~> 1.44"
    random = "~> 2.2"
  }
}

```

The `outputs.tf` file is also quite simple:

#### **Listing 5.12 outputs.tf**

```

output "website_url" {
  value = "https://${local.namespace}.azurewebsites.net/"
}

```

For your reference, the complete code from `main.tf` is shown below.

#### **Listing 5.13 complete code for main.tf**

```

resource "random_string" "rand" {
  length = 24
  special = false
  upper  = false
}

locals {
  namespace = substr(join("-", [var.namespace, random_string.rand.result])), 0, 24)
}

```

```

resource "azurerm_resource_group" "default" {
  name    = local.namespace
  location = var.region
}

resource "azurerm_storage_account" "storage_account" {
  name          = random_string.rand.result
  resource_group_name = azurerm_resource_group.default.name
  location      = azurerm_resource_group.default.location
  account_tier   = "Standard"
  account_replication_type = "LRS"
}

resource "azurerm_storage_container" "storage_container" {
  name        = "serverless"
  storage_account_name = azurerm_storage_account.storage_account.name
  container_access_type = "private"
}

module "ballroom" {
  source = "scottwinkler/ballroom/azure"
}

resource "azurerm_storage_blob" "storage_blob" {
  name          = "server.zip"
  storage_account_name = azurerm_storage_account.storage_account.name
  storage_container_name = azurerm_storage_container.storage_container.name
  type         = "block"
  source       = module.ballroom.output_path
}

data "azurerm_storage_account_sas" "storage_sas" {
  connection_string = azurerm_storage_account.storage_account.primary_connection_string

  resource_types {
    service = false
    container = false
    object = true
  }

  services {
    blob = true
    queue = false
    table = false
    file = false
  }

  start = "2016-06-19T00:00:00Z"
  expiry = "2048-06-19T00:00:00Z"

  permissions {
    read = true
    write = false
    delete = false
    list = false
    add = false
  }
}

```

```

create = false
update = false
process = false
}
}

locals {
package_url =
    replace("https://${azurerm_storage_account.storage_account.name}.blob.core.windows.net/${azurerm_storage_c
ontainer.storage_container.name}/${azurerm_storage_blob.storage_blob.name}${data.azurestorage_account_s
as.storage_sas.sas}", "%3d", "=")
}

resource "azurerm_app_service_plan" "plan" {
name      = local.namespace
location   = azurerm_resource_group.default.location
resource_group_name = azurerm_resource_group.default.name
kind      = "functionapp"
sku {
tier = "Dynamic"
size = "Y1"
}
}

resource "azurerm_application_insights" "application_insights" {
name      = local.namespace
location   = azurerm_resource_group.default.location
resource_group_name = azurerm_resource_group.default.name
application_type = "Web"
}

resource "azurerm_function_app" "function" {
name      = local.namespace
location   = azurerm_resource_group.default.location
resource_group_name = azurerm_resource_group.default.name
app_service_plan_id = azurerm_app_service_plan.plan.id
https_only = true
storage_connection_string = azurerm_storage_account.storage_account.primary_connection_string
version     = "~2"
app_settings = {
FUNCTIONS_WORKER_RUNTIME = "node"
WEBSITE_RUN_FROM_PACKAGE = local.package_url
WEBSITE_NODE_DEFAULT_VERSION = "10.14.1"
APPINSIGHTS_INSTRUMENTATIONKEY = azurerm_application_insights.application_insights.instrumentation_key
TABLES_CONNECTION_STRING = data.azurestorage_account_sas.storage_sas.connection_string
AzureWebJobsDisableHomepage = true
}
}
}

```

**NOTE** Some people like declaring local values all together at the top of the file, but I prefer declaring local values next to the resources that use them. Either approach is valid.

## 5.4 Deploying to Azure

We are done with the four steps required to set up the Azure serverless project and are ready to deploy! Run a `terraform init` and `terraform plan` to initialize Terraform and verify that the configuration code is correct.

```
$ terraform init && terraform plan
...
# azurerm_storage_container.storage_container will be created
+ resource "azurerm_storage_container" "storage_container" {
  + container_access_type = "private"
  + id        = (known after apply)
  + name      = "serverless"
  + properties = (known after apply)
  + resource_group_name = "ballroominaction"
  + storage_account_name = (known after apply)
}

# random_string.rand will be created
+ resource "random_string" "rand" {
  + id      = (known after apply)
  + length  = 24
  + lower   = true
  + min_lower = 0
  + min_numeric = 0
  + min_special = 0
  + min_upper = 0
  + number   = true
  + result   = (known after apply)
  + special  = false
  + upper    = false
}
```

Plan: 8 to add, 0 to change, 0 to destroy.

---

Note: You didn't specify an "-out" parameter to save this plan, so Terraform can't guarantee that exactly these actions will be performed if "terraform apply" is subsequently run.

Next deploy with a `terraform apply`. The command and subsequent output are as follow:

**WARNING!** You should probably run a `terraform plan` first. I do a `terraform apply -auto-approve` only to save space. It can be risky.

```
$ terraform apply -auto-approve
...
azurerm_function_app.function: Still creating... [10s elapsed]
azurerm_function_app.function: Still creating... [20s elapsed]
azurerm_function_app.function: Still creating... [30s elapsed]
azurerm_function_app.function: Still creating... [40s elapsed]
azurerm_function_app.function: Creation complete after 48s [id=/subscriptions/7deeca5c-dc46-45c0-8c4c-7c3068de3f63/resourceGroups/ballroominaction/providers/Microsoft.Web/sites/ballroominaction-23sr1wf]
```

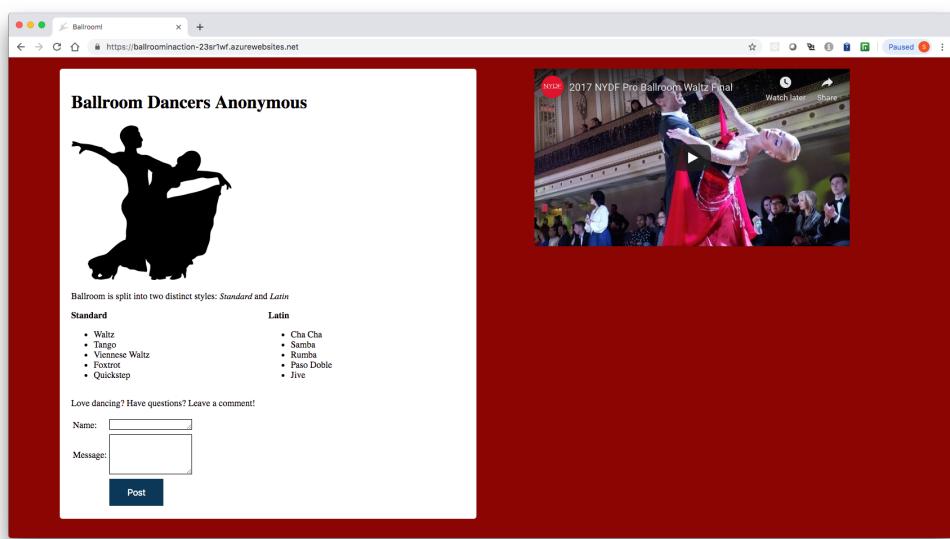
```
Apply complete! Resources: 8 added, 0 changed, 0 destroyed.
```

Outputs:

```
website_url = https://ballroominaction-23sr1wf.azurewebsites.net/#A
```

#A This is a live link if you just want to view the final result

You can navigate to the deployed website in the browser. Figure 5.15 shows what this will look like.



**Figure 5.15 Deployed Ballroom Dancers Anonymous website**

**TIP** It's surprisingly hard to find simple examples for Azure serverless projects, so I've intentionally made the source code minimalistic and easy to understand. Feel free to peruse my work or use it as a template for your own serverless projects. You can find it either on [Github<sup>4</sup>](#), or in the .terraform/modules/ballroom directory.

Don't forget to do a `terraform destroy` to cleanup! This will take down all the infrastructure that we provisioned in Azure.

<sup>4</sup> <https://github.com/scottwinkler/terraform-azure-ballroom>

```
$ terraform destroy -auto-approve
...
azurerm_resource_group.default: Still destroying... [id=/subscriptions/7deeca5c-dc46-45c0-8c4c-
    ...de3f63/resourceGroups/ballroominaction, 1m30s elapsed]
azurerm_resource_group.default: Still destroying... [id=/subscriptions/7deeca5c-dc46-45c0-8c4c-
    ...de3f63/resourceGroups/ballroominaction, 1m40s elapsed]
azurerm_resource_group.default: Destruction complete after 1m48s

Destroy complete! Resources: 8 destroyed.
```

## 5.5 Combining Azure Resource Manager (ARM) with Terraform

Azure Resource Manager (ARM) is Microsoft's own Infrastructure as Code technology that allows you to provision resources to Azure using JSON configuration files. If you've ever used AWS CloudFormation or GCP Deployment Manager, it's a lot like that, so most of the concepts from this section will carry over to these technologies as well. Nowadays, Microsoft is heavily promoting Terraform over Resource Manager, but there are still some legacy use cases of ARM. The three cases where I found ARM to be useful were:

1. Deploying resources that aren't yet supported by Terraform
2. Migrating legacy ARM code to Terraform
3. Generating configuration code

### 5.5.1 Deploying Unsupported Resources

Back in ye olden days, when Terraform was still an emerging technology, Terraform providers didn't enjoy the same level of support they have today (even for the major clouds). In Azure's case, there were many resources that were unsupported by Terraform long after their general availability (GA) release. For example, Azure IoT Hub was announced GA in 2016, but it took more than two years until the resource was finally added to the Azure Provider! In that awkward gap period, if you wished to deploy an IoT Hub from Terraform, your best bet was to deploy an ARM template from Terraform (see Snippet 5.1). This was a way of "bridging the gap", so to speak, between what was possible with Terraform and what was possible with ARM. The same was true for some unsupported resources in AWS and GCP using Cloud Formation and GCP Deployment Manager.

#### **Snippet 5.1 Deploying an ARM deployment with Terraform**

```
resource "azurerm_template_deployment" "template_deployment" {
  name      = "terraform-ARM-deployment"
  resource_group_name = azurerm_resource_group.resource_group.name
  template_body   = file("${path.module}/templates/iot.json")
  deployment_mode  = "Incremental"

  parameters = {
    IoTHubs_my_iot_hub_name = "ghetto-hub"
  }
}
```

As Terraform has matured, provider support has swelled to encompass more and more resources, and today you'd be hard pressed to find a resource that isn't natively supported by Terraform. Regardless, there are still some occasional situations where deploying ARM templates from Terraform could still be a viable strategy (even if it is natively supported by Terraform). For example, if there are some Terraform resources that are poorly implemented, or are otherwise incomplete and don't have all the features that the corresponding ARM template has, you may be better off deploying an ARM template with Terraform.

### 5.5.2 Migrating from Legacy Code with the Strangler Façade Pattern

The strangler façade pattern is a pattern for migrating from a legacy system to a new system by slowly replacing the legacy parts with new parts until the old system is completely superseded by the new system. At that point, the old system may be safely decommissioned. It's called the strangler façade pattern because the new system is said to "strangle" the legacy system until it dies off entirely (see figure 5.16). You've probably encountered something exactly like this as it's a fairly common strategy, especially for updating APIs and services which must uphold a certain service level agreement (SLA).

How this applies to Terraform is that oftentimes there will be legacy code written in ARM or CloudFormation, that needs to be migrated to Terraform without causing downtime to the service. These legacy templates cannot easily be converted to Terraform right away, so the best approach is to adopt the strangler façade pattern. You start by throwing all the legacy code into an `azurerm_template_deployment` or `aws_cloudformation_stack` resource. Over time, you incrementally replace specific functionality from the old ARM or CloudFormation Stack with native Terraform resources until you are entirely in Terraform (where everything is sunshine and rainbows, of course).

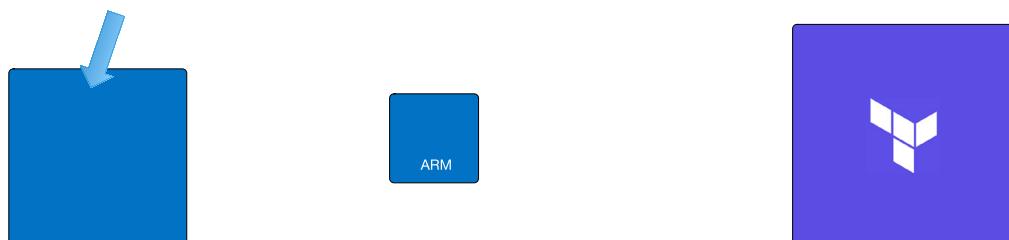


Figure 5.16 The Strangler facade pattern for migrating ARM to Terraform. At first you start off with a huge ARM template wrapped with an `azurerm_template_deployment` resource and not much else. Over time, resources are taken out of the ARM template and configured as native Terraform resources. Eventually, you no longer need the ARM template because everything is now a managed Terraform resource

### 5.5.3 Generating Configuration Code

This last point is unique to Azure and is without question my favorite feature about ARM; after all, who doesn't like generated code? I'll preface this by saying that the most painful thing about Terraform is that it takes a lot of work to translate what you want into configuration code<sup>5</sup>. It's usually much easier to point and click around the console, until you have what you want, then export that as a template. This is exactly what Azure resource groups let you do. You can take any resource group that is currently deployed, export it as an ARM template file, and then deploy that template with Terraform (see figure 5.17).

```

{
  "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
  "contentVersion": "1.0.0.0",
  "parameters": {
    "sites_ballroomaction_23rlwf_name": {
      "defaultValue": "ballroomaction-23rlwf",
      "type": "String"
    },
    "serverFarms_ballroomaction_23rlwf_name": {
      "defaultValue": "ballroomaction-23rlwf",
      "type": "String"
    },
    "components_ballroomaction_23rlwf_name": {
      "defaultValue": "ballroomaction-23rlwf",
      "type": "String"
    },
    "storageAccounts_23rlwfmhobedlnrvvly_name": {
      "defaultValue": "23rlwfmhobedlnrvvly",
      "type": "String"
    },
    "actionGroups_Application20Insights20ear%20detection_name": {
      "defaultValue": "Application20Insights20ear%20detection",
      "type": "String"
    }
  },
  "variables": {},
  "resources": [
    {
      "type": "microsoft.insights/actionGroups"
    }
  ]
}

```

**Figure 5.17** You can take any resource group that is currently deployed, export it as an ARM template file, and then deploy that template with Terraform

**WARNING** Generated ARM templates are not always a 1:1 mapping of what is currently deployed in a resource group. You should refer to the Azure ARM documentation for a definitive reference on what is and what is not currently supported<sup>6</sup>

<sup>5</sup> There are a number of open source projects that aim to address this problem, most notably “Terraforming”, located here: <https://github.com/dtan4/terraform>. HashiCorp also promises that they will improve imports to natively support generating configuration code from deployed resources in some future release of Terraform.

<sup>6</sup> <https://docs.microsoft.com/en-us/azure/templates/>

The beauty (or curse) of this approach is that you can sketch up your entire project in the console and deploy it via Terraform without having to write any configuration code (except some small amount of boilerplate). Sometime in the future, if you care enough, you could migrate the quick and dirty template to native Terraform using the Strangler Façade pattern mentioned in the previous section. I like to think of this trick as a form of rapid prototyping.

### **The Dark Road of Generated Code**

Besides Azure Resource Manager, there are a variety of tools that promise generating configuration code through some means or other. If you find yourself contemplating a burning desire to generate configuration code, I would highly recommend you consider the use of Terraform modules instead. Modules are the recommended vehicle for code reuse in Terraform and can be extremely versatile when using features such as dynamic blocks and for expressions.

In my opinion, writing Terraform code is the easy part, it's figuring out what you want to do that's hard. Generated code has a high "coolness" factor associated with it, but I believe it will only be of limited use at best, especially because complex automation and code generation tools have a tendency to lag behind the latest version of whatever technology they are tailored for.

I'd also like to remind you that just because services like Wordpress, Wix, and Squarespace allow non-technical people to create websites doesn't mean we've eliminated the need for quality frontend JavaScript developers. It's the same for Terraform. These tools that allow you to generate code should be thought of as potentially useful ways to augment your productivity, rather than eliminating the need to know how to write clean Terraform code yourself.

## **5.6 Summary**

- Azure (and other clouds) provide simple, cost effective serverless offerings. Terraform can be used to easily deploy an all-in-one serverless application to Azure.
- Approach problem solving with Terraform like any other problem-solving exercise: the more planning you can do ahead of time, the better. An effective problem solving strategy involves: 1) defining the problem 2) doing research 3) choosing the tools 4) building a prototype 5) developing the final product.
- Application source code and other build artifacts can be stored in the same directory as your infrastructure code, but they do not have to be. One clever (albeit unconventional) method to is to use modules as a vehicle to download these files during `terraform init`.
- Terraform providers are mostly written by hand and are not without their flaws. Sometimes you need to be creative to get your code to work; filing issues on GitHub repos will only get you so far.
- Azure Resource Manager (ARM) is an interesting technology that can be combined with Terraform to potentially patch holes in Terraform, or even allow you to skip writing Terraform entirely. Use with caution, however, because it's not a panacea.

# 6

## *Terraform with Friends*

### **This chapter covers:**

- Developing an S3 remote backend module
- Comparing Flat vs. Nested Module Structures
- Publishing modules via GitHub and the Terraform Module Registry
- Switching between workspaces with the greatest of ease
- Examining Terraform Cloud and Terraform Enterprise

Like it or not, software development is a team sport. At some point, you'll need to collaborate with friends and coworkers. Previously, we've saved state to our local machine, which is fine for individual contributors, but not for team environments. Suppose Sally from SRE wants to update the configuration code of a resource and redeploy. Unless she had a copy of the existing state file, there is no way she could accomplish this herself. You could give her the state file or check it into a version-controlled source repository, but this would unnecessarily expose secrets and wouldn't address the problem of data synchronization. Data synchronization problems happens when two people are trying to `terraform apply` at the same time, which leads to a race condition and potentially corrupted state files.

Terraform with friends is all about enabling multiple users to access the same state files without stepping on anyone's toes. The best way to accomplish this is with *remote backends* and *workspaces*. In this chapter, we'll develop a production ready S3 remote backend module, publish it on GitHub and the Terraform Module Registry. Next, we'll deploy the backend and test it out. Along the way, we'll cover workspaces and how they can reuse the same configuration code to deploy to multiple environments. Finally, we'll introduce HashiCorp's proprietary products that solve the same problem: Terraform Cloud and Terraform Enterprise.

## 6.1 Standard and Enhanced Backends

A *backend* in Terraform determines how state is loaded and how operations like `terraform plan` and `terraform apply` are executed. We've actually been using a *local backend* this whole time, because that's the default behavior of Terraform. Besides storing state, backends can also be configured for:

1. Synchronizing access to state files via locking
2. Storing sensitive information securely
3. Keeping a history of all state file revisions
4. Determining how operations are implemented

Some backends have the ability to completely overhaul the way Terraform works, but most are not much different than the local backend. The main responsibility of a backend is to determine how state files are stored and accessed, so it's unsurprising that they basically all work the same way: state files are stored in some location and there's a built-in way to synchronize access to the state file. Most backends support encryption at rest and versioning of state files, but you'll have to refer to the specific backend documentation to learn what is supported and what isn't.<sup>1</sup>

Besides local backends, there're also *enhanced* and *standard backends*. Enhanced backends are relatively new and allow you to do sophisticated things like run CLI operations on a remote machine and stream the results back to your local terminal. They also allow you to read variables and environment variables stored on the remote machine, so there's no longer a need for a variables definition (`terraform.tfvars`) file. Although enhanced backends are great, many of the best features are reserved for customers of Terraform Cloud and Terraform Enterprise. Don't worry though, most people who use Terraform – even at scale – will be perfectly content with one of the standard backend implementations.

The most popular standard backend is the S3 remote backend for AWS (probably because most people use AWS). In the next few sections, I'll show you how to build and deploy an S3 backend module, as well as the workflow for deploying against it. Once you get it set up, you'll no doubt wonder how you ever made do without. A basic diagram of how the S3 backend works is shown in figure 6.1.

<sup>1</sup> <https://www.terraform.io/docs/backends/types/index.html>

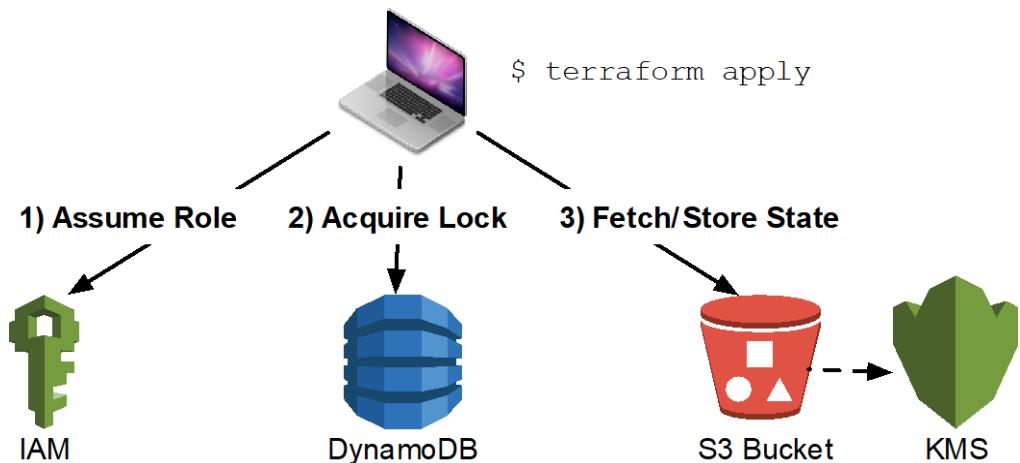


Figure 6.1 Every Terraform client that shares the same S3 backend will fetch and store their state from a common S3 bucket. The state file itself is encrypted at rest using KMS. Access is controlled by a least privileged IAM policy and synchronized with a DynamoDB locking mechanism.

## 6.2 Developing an S3 Backend Module

Our goal is to architect a module that can eventually be used to deploy a production ready S3 backend. This backend can be used not only by yourself, or your team, but by other teams as well. If your primary cloud is Azure or GCP, then the code here will not be immediately useful, but the idea is still the same. Since standard backends are more similar than dissimilar, you can apply what you learn here to develop a custom solution for whichever backend you prefer.

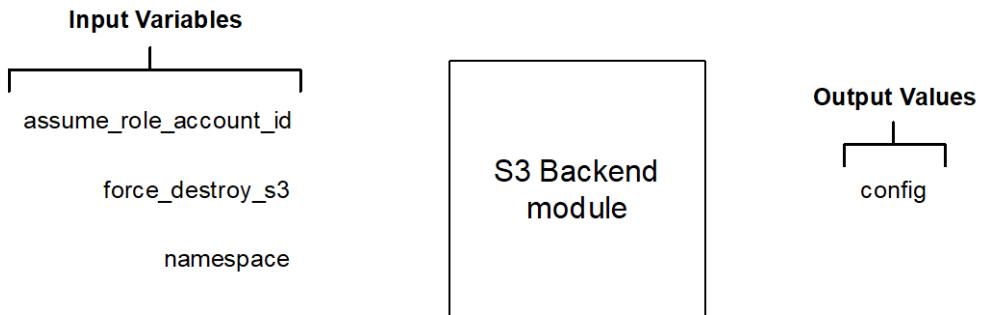
This whole project was designed from the exacting requirements laid out in the official documentation<sup>2</sup>, which does a great job explaining *what* you need to do, but not *how* to do it. It's like when you want to build a computer, buying the parts is not enough, you still need to know how to assemble them. Since you're probably going to want to deploy an S3 backend anyways, I hope to save you some time and trouble. Also, as we're developing a module that can be published on the Terraform Module Registry, you'll learn the skills necessary to share your modules with others.

### 6.2.1 Architecture

I always like to start off by considering the overall inputs and outputs from a black box perspective. There are three input variables (for configuring various settings that we'll talk more

<sup>2</sup> <https://www.terraform.io/docs/backends/types/s3.html>

about soon) and one output value (that has all the information required for other workspaces to initialize themselves against the S3 backend). This is depicted by figure 6.2.



**Figure 6.2** There are three inputs and one output for the S3 backend module. The output value “config” has all the information required for a workspace to initialize itself against the S3 backend

Considering what's inside the box, there are four distinct components required to deploy an S3 backend. These are:

1. DynamoDB table (for locking)
2. S3 Bucket and KMS Key (for state storage and encryption at rest)
3. IAM Least Privileged Role (so other AWS accounts can assume a role to this account and perform deployments against the S3 backend)
4. Miscellaneous housekeeping resources (we'll talk about these more later)

I've drawn a helpful diagram to help you visualize the relationship from a Terraform dependency perspective (see figure 6.3). As you can see, there are four discernible islands of resources that don't have a clear dependency relationship on each other. These components would be excellent candidates for modularization, as discussed in chapter 4, but I won't advocate doing that here as it would be a bit overkill. Instead, I'll introduce a different design pattern for organizing code that's perfect for this situation. Although popular, it doesn't actually have a colloquial name, so I'll refer to it simply as a *flat module structure*.

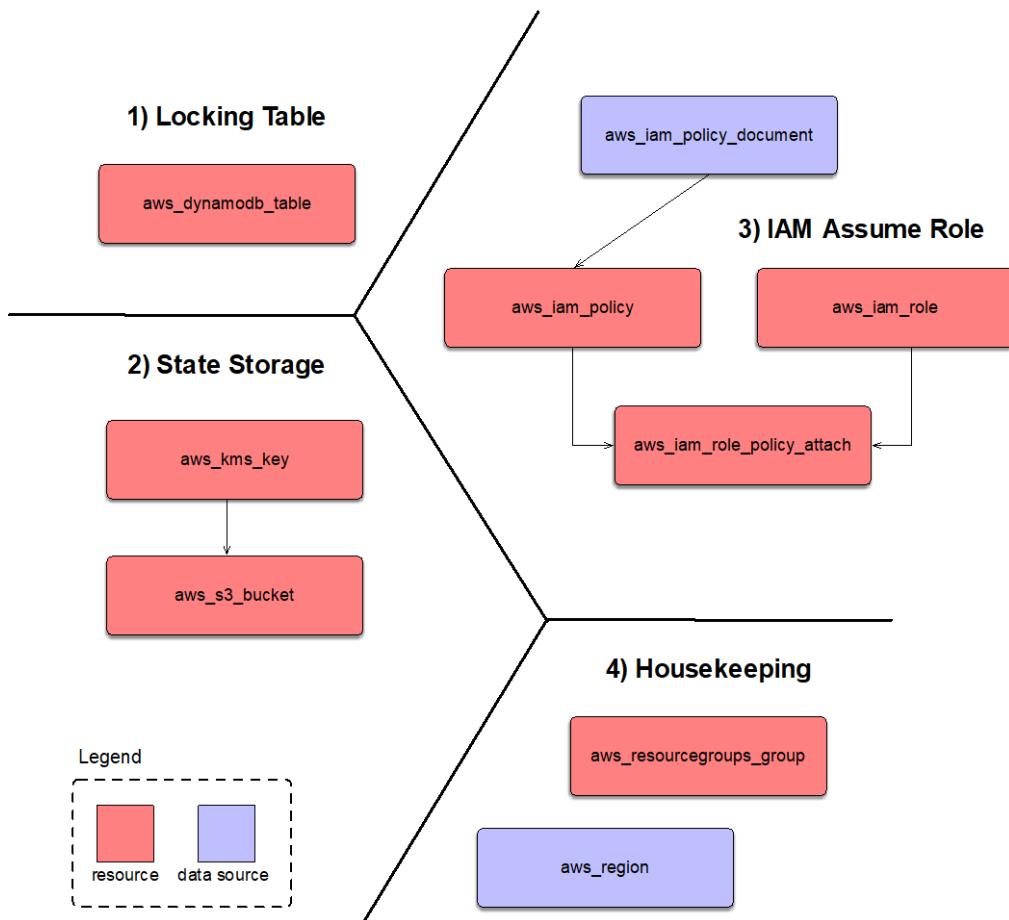
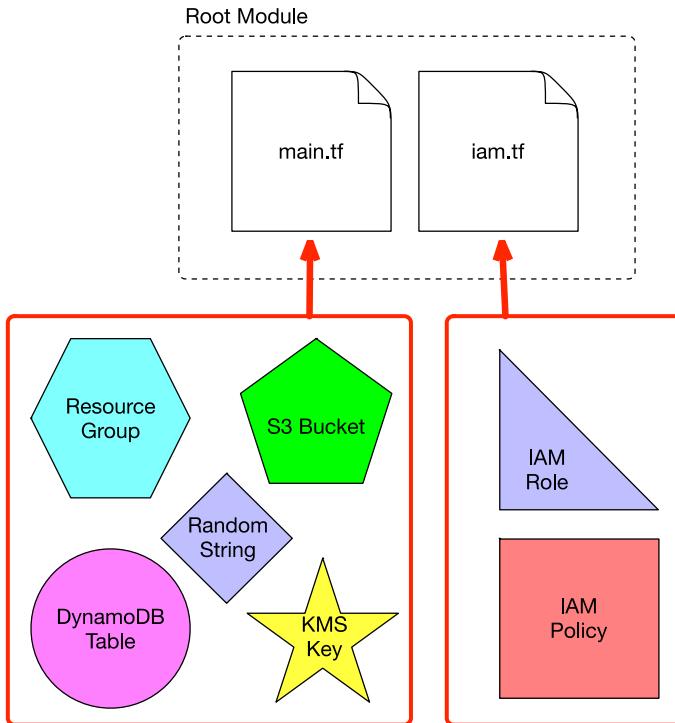


Figure 6.3 Detailed architecture diagram showing the four distinct components that make up this module

### 6.2.2 Flat Module Structure

*Flat modules* (as opposed to *nested modules*) are when you organize your code by creating lots of little `.tf` files within a single monolithic module. Each file in the module contains all the code for deploying an individual component, which would otherwise be broken out into its own module. The primary advantage of flat module structures over nested modules is reduced boilerplate, and easier codebase navigation. For example, instead of creating a fully-fledged module for IAM resources, that code could be put into a file name `iam.tf`. This is illustrated by figure 6.4.



**Figure 6.4** A flat module structure applied to the S3 backend module. All IAM resources go in `iam.tf`, and everything else goes in `main.tf`

For this particular scenario, it actually makes a lot of sense to do it this way: the IAM resources code is slightly too long to be included in `main.tf`, and yet not quite long enough to warrant being a separate module.

**TIP** There's no fixed rule about how long the configuration code in a single file can be, but I find it best to stick to a ~200 lines maximum rule of thumb

**WARNING** Think carefully before deciding to use a flat module structure for code organization. This pattern permits a high degree of inter-component coupling, which can make your code difficult to navigate and understand.

### Flat vs Nested Module Structures

Flat module structures are most effective in small-to-medium sized codebases, and only when your code can be cleanly subdivided into components that are functionally independent of each other (i.e. not having dependencies on resources declared in other files). Nested module structures, on the other hand, are much more useful for large, complex, or shared codebases.

As a concrete analogy, flat module structures are like storing stateful information in global variables, whereas nested module structures are like using classes. Global variables can be convenient and make your code more compact and quicker to write, but they can also make your code harder to understand and reason about. Likewise, dividing code into classes can be more upfront boilerplate, but the added readability is usually worth it to avoid tightly coupled architectures. Always understand the tradeoffs between different design patterns when deciding how to organize your project structure.

### 6.2.3 Writing the Code

Without further ado, let's dive into the thick of things. Start by creating 6 files: `variables.tf`, `main.tf`, `iam.tf`, `outputs.tf`, `versions.tf` and `README.md`. Listing 6.1 shows the code for `variables.tf`.

**NOTE** We don't need a `providers.tf` because this is a module. All providers will be implicitly passed in by the root module during initialization.

#### Listing 6.1 `variables.tf`

```
variable "namespace" {
  description = "The project namespace to use for unique resource naming"
  default     = "s3backend"
  type        = string
}

variable "principal_arn" {
  description = "AWS principal arn allowed to assume the IAM role"
  default     = null
  type        = string
}

variable "force_destroy_state" {
  description = "Force destroy the s3 bucket containing state files?"
  default     = true
  type        = bool
}
```

The complete code for provisioning the S3 bucket, KMS key, and DynamoDB table is shown in Listing 6.2. I put all this in `main.tf` because they're the most important resources of the module, and because this is the first file most people will look at when reading through your project. The key to enabling developer productivity is having a well organized codebase, so others can easily find what they are looking for even if they are not Terraform experts.

#### Listing 6.2 `main.tf`

```
data "aws_region" "current" {} #A

resource "random_string" "rand" {
  length = 24
```

```

special = false
upper  = false
}

locals {
  namespace = substr(join("-", [var.namespace, random_string.rand.result]), 0, 24)
}

resource "aws_resourcegroups_group" "resourcegroups_group" {
  name = "${local.namespace}-group"

  resource_query { #B
    query = <<-JSON
{
  "ResourceTypeFilters": [
    "AWS::AllSupported"
  ],
  "TagFilters": [
    {
      "Key": "ResourceGroup",
      "Values": ["${local.namespace}"]
    }
  ]
}
  JSON
}
}

resource "aws_kms_key" "kms_key" {
  tags = {
    ResourceGroup = local.namespace
  }
}

resource "aws_s3_bucket" "s3_bucket" {
  bucket = "${local.namespace}-state-bucket"
  force_destroy = var.force_destroy_state

  versioning {
    enabled = true
  }

  server_side_encryption_configuration {
    rule {
      apply_server_side_encryption_by_default {
        sse_algorithm  = "aws:kms"
        kms_master_key_id = aws_kms_key.kms_key.arn
      }
    }
  }

  tags = {
    ResourceGroup = local.namespace
  }
}

resource "aws_dynamodb_table" "dynamodb_table" {

```

```

name      = "${local.namespace}-state-lock"
hash_key  = "LockID"
billing_mode = "PAY_PER_REQUEST" #C
attribute {
  name = "LockID"
  type = "S"
}
tags = {
  ResourceGroup = local.namespace
}
}

```

#A This will be used to set an output value  
#B Populate the resource group based on a tagging schema  
#C Provision a serverless database for state locking

Next is the code for `iam.tf`. This particular code creates a least privileged IAM role that another AWS account can assume to deploy against the S3 remote backend. To clarify: you would be expected to have a single dedicated AWS account for housing all S3 remote backends that your company manages, and then have other AWS accounts assume a role into this account for least privileged access. Deployment users would only have privileges to utilize the S3 backend/state files they are authorized for. In this example, the IAM role we declare grants permissions to store an S3 object from a specific S3 bucket, and get/delete records from the DynamoDB table.

**NOTE** The purpose of having multiple AWS accounts assume a least privileged IAM role into another account is to prevent users from unauthorized access to state files. Some state files have important secrets stored in plain text, which shouldn't be read by just anyone.

### Listing 6.3 iam.tf

```

data "aws_caller_identity" "current" {}

locals {
  principal_arn = var.principal_arn != null ? var.principal_arn : data.aws_caller_identity.current.arn #A
}

resource "aws_iam_role" "iam_role" {
  name = "${local.namespace}-tf-assume-role"

  assume_role_policy = <<-EOF
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": "sts:AssumeRole",
      "Principal": {
        "AWS": "${local.principal_arn}"
      },
      "Effect": "Allow"
    }
  ]
}

```

```

EOF

tags = {
  ResourceGroup = local.namespace
}
}

data "aws_iam_policy_document" "policy_doc" { #B
  statement {
    actions = [
      "s3>ListBucket",
    ]
  }

  resources = [
    aws_s3_bucket.s3_bucket.arn
  ]
}

statement {
  actions = ["s3GetObject", "s3PutObject"]

  resources = [
    "${aws_s3_bucket.s3_bucket.arn}/*",
  ]
}

statement {
  actions = [
    "dynamodb:GetItem",
    "dynamodb:PutItem",
    "dynamodb:DeleteItem"
  ]
  resources = [aws_dynamodb_table.dynamodb_table.arn]
}
}

resource "aws_iam_policy" "iam_policy" {
  name = "${local.namespace}-tf-policy"
  path = "/"
  policy = data.aws_iam_policy_document.policy_doc.json
}

resource "aws_iam_role_policy_attachment" "policy_attach" {
  role = aws_iam_role.iam_role.name
  policy_arn = aws_iam_policy.iam_policy.arn
}
}

```

#A If no principal arn was specified, use the arn of the account that deployed this module  
#B Least privileged policy to attach to role

There are three pieces of information that a workspace will need to be able to initialize and deploy against the S3 backend. These are:

1. The name of the bucket
2. The region the S3 backend is deployed to

### 3. The arn of the role that can be assumed

Since this isn't the root module, the outputs will need to be bubbled up in order to be visible after a `terraform apply` (we'll do this later). Outputs are shown below, in Listing 6.4.

#### **Listing 6.4 outputs.tf**

```
output "config" {
  value = {
    bucket      = aws_s3_bucket.s3_bucket.bucket
    region      = data.aws_region.current.name
    role_arn    = aws_iam_role.iam_role.arn
    dynamodb_table = aws_dynamodb_table.dynamodb_table.name
  }
}
```

Even though we don't declare providers, it's still a good idea to version lock modules, so that anyone consuming the module will know whether they are using compatible providers or not.

#### **Listing 6.5 versions.tf**

```
terraform {
  required_version = "~> 0.12"
  required_providers {
    aws = "~> 2.19"
    random = "~> 2.1"
  }
}
```

Next we need to create a `README.md` file. Believe it or not, having a `README.md` is actually a requirement for registering a module with the Terraform Module Registry. You've got to hand it to HashiCorp for laying down the law on these sorts of things. Let's make a dirt simple `README.md` to make HashiCorp happy (see Listing 6.6).

**TIP** there's a neat open source tool called `terraform-docs`<sup>3</sup> that automatically generates documentation from your configuration code

#### **Listing 6.6 README.md**

```
# S3 Backend Module
This module will deploy an S3 remote backend for Terraform #A
```

#A You'll probably want to write more documentation than this for your own modules – such as what the inputs and outputs are, and how to use it.

<sup>3</sup> <https://github.com/segmentio/terraform-docs>

Finally, since we'll first be uploading this to a GitHub repo, you'll want to create a `.gitignore` file. A pretty typical one for Terraform modules is shown in Listing 6.7.

#### **Listing 6.7 .gitignore**

```
.DS_Store  
.vscode  
*.tfstate  
*.tfstate.*  
terraform  
**/terraform/*  
crash.log
```

### **6.3 Sharing Modules**

Great, so now we have a module. But how do we share it with friends and coworkers? Although there is a Terraform Module Registry, which I personally think is the best option, there's actually a number of possible avenues for sourcing modules. The most common approach is to use GitHub repos, but I've also found S3 buckets to be a good option. In this section, I'll show you how to publish and source a module in two different ways: first from GitHub, then from the Terraform Module Registry.

**NOTE** You'll need to upload your code to GitHub even if you only wish to use the Terraform Module Registry, because the Terraform Module Registry sources from public GitHub repos

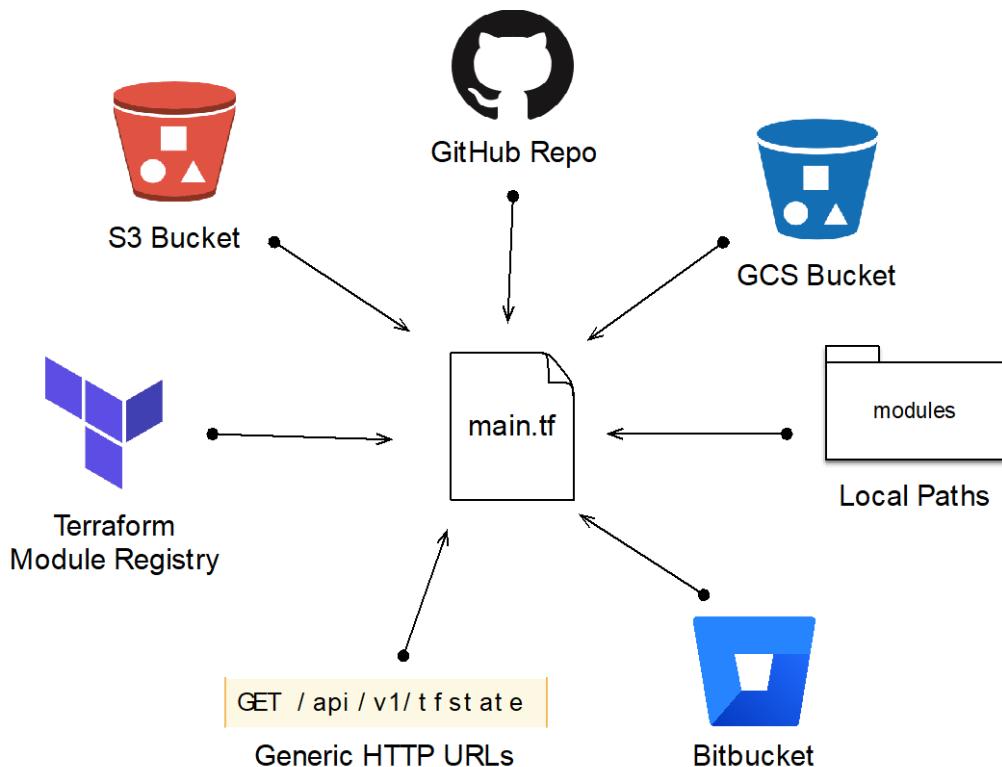


Figure 6.5 Modules can be sourced from multiple possible avenues, including: local paths, GitHub repos, and the Terraform Module Registry.

### 6.3.1 GitHub

Sourcing modules from GitHub is quite easy. Just create a repo with a name of the form: `terraform-<PROVIDER>-<NAME>`. There're no rules about what "provider" and "name" mean in the context of a module, but I typically think of "provider" as the cloud that I am deploying to, while "name" is a helpful descriptor of the project. Therefore, the particular module we are deploying might be named "terraform-aws-s3backend".

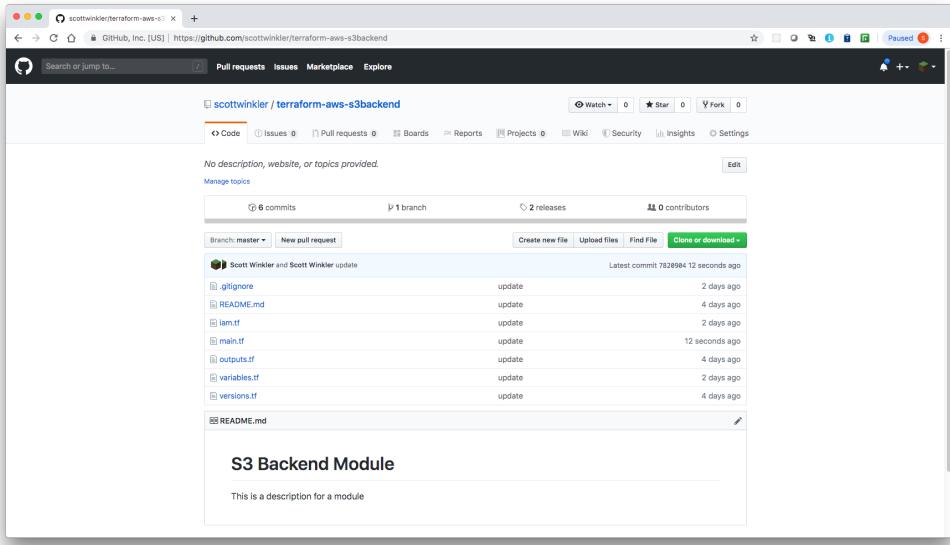


Figure 6.6 Example GitHub repo for `terraform-aws-s3backend` module

A sample configuration for sourcing a module from a GitHub repo is shown in Snippet 6.1.

#### Snippet 6.1 Sample configuration code

```
module "s3backend" {
  source = "github.com/scottwinkler/terraform-aws-s3backend"
}
```

**TIP** If you wish to source from a generic Git repo, you can do so by prefixing the address with `git:::`. Any valid Git URL can follow this prefix, allowing you to version control the module by specifying a branch or tag name.

### 6.3.2 Terraform Module Registry

The Terraform Module Registry is free and easy to use, all you need is a GitHub account to get started<sup>4</sup>. After you sign-in, it's just a few clicks in the UI to register a module so that other people can start using it. Because the Terraform Module Registry always reads from public GitHub repos, by publishing your module in the registry, you're making your module available to everyone. One of the perks of Terraform Enterprise is that it lets you have your own private

<sup>4</sup> The website is located here: <https://registry.terraform.io>

Terraform Module Registry, which can be useful for sharing private modules in large organizations.

**NOTE** As long as you aren't putting secrets in GitHub, there's no real reason to fear publishing your module on the Terraform Module Registry. Open source is a good thing.

The implementation of the Terraform Module Registry is not complicated in the least bit; the way I think about it is that it's little more than a glorified key-value store that maps source keys to GitHub repo locations. Its main benefit is that it enforces certain naming conventions and standards based on established best practices for publishing modules<sup>5</sup>. It also makes it easy to version control and search for other people's modules by name or provider. Here's a list of the official rules<sup>6</sup>:

1. Be a public repo on GitHub
2. Named of the form: `terraform-<PROVIDER>-<NAME>`
3. Have a `README.md` (preferably with some example usage code)
4. Follow the standard module structure (`main.tf`, `variables.tf`, `outputs.tf`)
5. Semantic versioned tags for releases (e.g. v0.1.0)

I would highly encourage you to try this yourself. In the following figures you can see just how easy this is to do. First create a tagged release in GitHub (figure 6.7). Next, sign-in to the Terraform Module Registry UI and click the "Publish" button (figure 6.8). At this point you'll need to select the GitHub repo you wish to publish (figure 6.9) and wait for it to be published (figure 6.10).

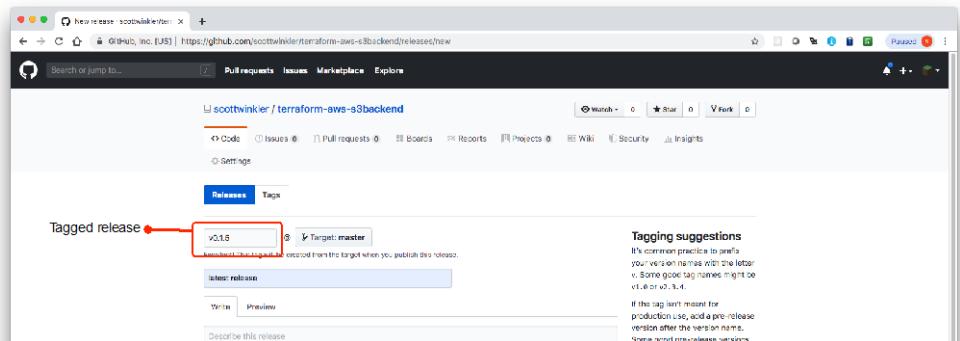


Figure 6.7 Create a tagged release in GitHub using semantic versioning

<sup>5</sup> HashiCorp's best practices for modules can be found here: <https://www.terraform.io/docs/modules/index.html>

<sup>6</sup> Official rules for the Terraform Module registry: <https://www.terraform.io/docs/registry/modules/publish.html>

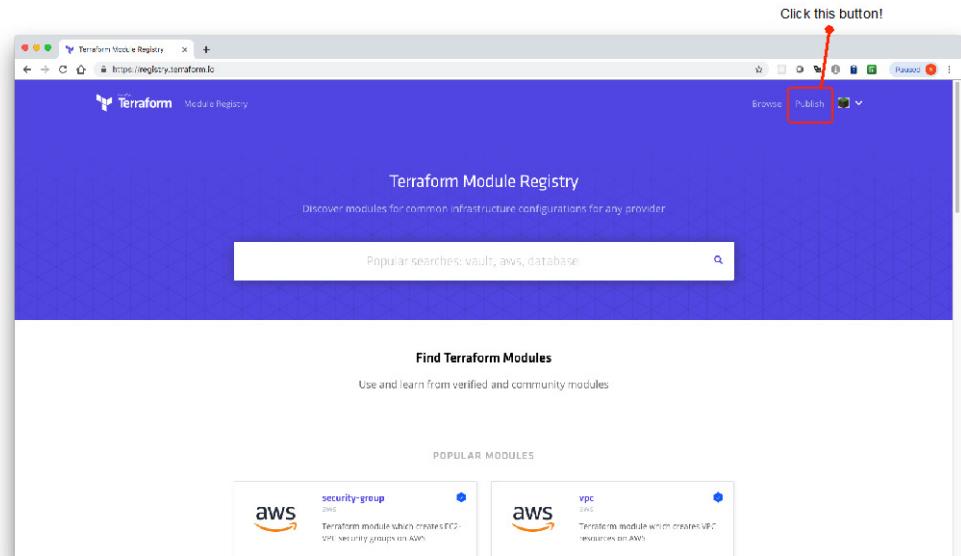


Figure 6.8 Navigate to Terraform Module Registry home page

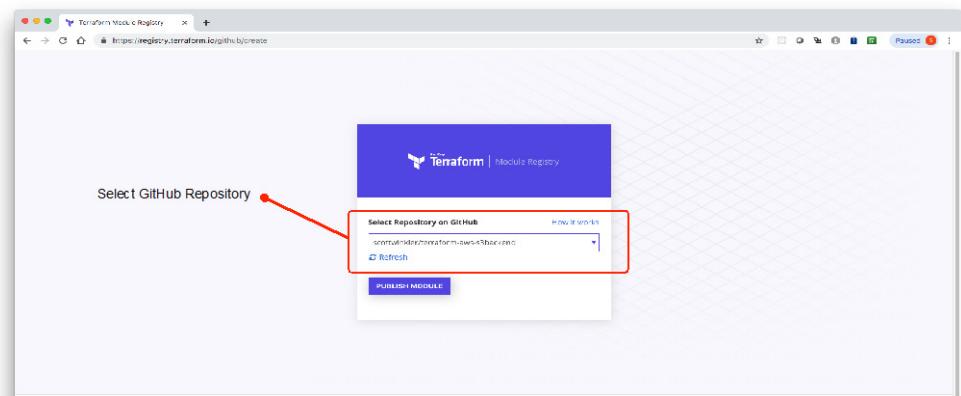


Figure 6.9 Choose a GitHub repo to register as a module

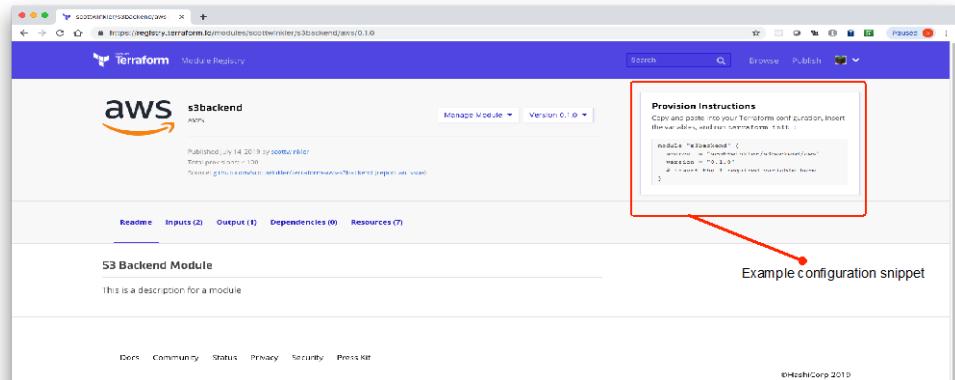


Figure 6.10 Published module in the Terraform Module Registry

## 6.4 Everyone Gets an S3 Backend!

Usually you want to have one S3 backend per team, where a team is eight or fewer people that work together on a regular basis (i.e. the two pizza rule). This strikes a good balance between granting people enough autonomy to do their job, while still minimizing security risk. Let's suppose, for the purpose of this scenario, that we need to deploy an S3 backend for a team of millennials called "Team Pokémon". After we deploy the S3 backend for them, we'll need to validate that it behaves normally before the members of Team Pokémon can be onboarded. As part of this validation process, we'll examine *workspaces*, and see how they can be used in conjunction with remote backends to deploy multiple environments from the same set of configuration code.

### 6.4.1 Deploying the S3 Backend

We need to create a harness for deploying the S3 backend module. Listing 6.8 shows the minimum code necessary to deploy this module. If you published the module on GitHub or the Terraform Module Registry, you can set the source of the module to point to your module, otherwise you can use mine that I've already published ahead of time for your convenience. Please create a new Terraform project with a file containing this code.

#### **Listing 6.8 s3backend.tf**

```

provider "aws" {
  region = "us-west-2" #A
}

module "s3backend" {
  source = "scottwinkler/s3backend/aws" #B
  namespace  = "pokemon"
}

```

```
output "s3backend_config" {
  value = module.s3backend.config
}
```

#A you can change the region if you want  
#B If you deployed the module on Terraform Module Registry, update the source to point to your module instead of mine

**TIP** You can copy the s3backend module to deploy multiple S3 backends at once (e.g. one per team). This is useful for limiting the blast radius in the event of a security breach. S3 backends are relatively inexpensive, so feel free to provision as many as you wish.

Start by running a `terraform init` followed by a `terraform apply`

```
$ terraform init && terraform apply
...
# module.s3backend.random_string.rand will be created
+ resource "random_string" "rand" {
  + id      = (known after apply)
  + length  = 8
  + lower   = true
  + min_lower = 0
  + min_numeric = 0
  + min_special = 0
  + min_upper = 0
  + number   = true
  + result   = (known after apply)
  + special  = false
  + upper    = false
}
```

Plan: 8 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?  
Terraform will perform the actions described above.  
Only 'yes' will be accepted to approve.

Enter a value:

When you're ready, confirm and wait for the resources to be provisioned:

```
...
module.s3backend.aws_iam_policy.iam_policy: Creation complete after 1s [id=arn:aws:iam::215974853022:policy/tf-policy]
module.s3backend.aws_iam_role_policy_attachment.policy_attach: Creating...
module.s3backend.aws_iam_role_policy_attachment.policy_attach: Creation complete after 1s [id=tf-assume-role-20190722062228664100000001]
```

Apply complete! Resources: 8 added, 0 changed, 0 destroyed.

Outputs:

```
s3backend_config = {
  "bucket" = "pokemon-q56ylfpq6bzrw3dl-state-bucket"
  "dynamodb_table" = "pokemon-q56ylfpq6bzrw3dl-state-lock"
  "region" = "us-west-2"
```

```
"role_arn" = "arn:aws:iam::215974853022:role/pokemon-q56ylfpq6bzrw3dl-tf-assume-role"
}
```

Save the output, you'll need it for the next step.

#### 6.4.2 Storing State in the S3 Backend

Now we're ready for the interesting part: validating that the S3 backend actually works. Create a new directory somewhere outside of the previous project with a `test.tf` file and configure the backend using the output from the previous section (see Listing 6.9). Since this is a test for Team Pokémon, we'll continue with the Pokémon theme by setting the key attribute to: "pikachu/thunderbolt". This key is used to determine the path to store the state file in the S3 bucket.

##### **Listing 6.9** `test.tf`

```
terraform {
  backend "s3" { #A
    bucket = "pokemon-q56ylfpq6bzrw3dl-state-bucket"
    key    = "pikachu/thunderbolt" #B
    region = "us-west-2"
    encrypt = true #C
    role_arn = "arn:aws:iam::215974853022:role/pokemon-q56ylfpq6bzrw3dl-tf-assume-role"
    dynamodb_table = "pokemon-q56ylfpq6bzrw3dl-state-lock"
  }
  required_version = "~> 0.12"
  required_providers {
    null = "~> 2.1"
  }
}
```

#A the backend is configured in the terraform settings block

#B you should set the key with a prefix (i.e. "pikachu") to allow for storing multiple projects within the same bucket

#C enables server side encryption of the state file

**NOTE** You need AWS credentials to assume the role specified by the backend `role_arn` attribute. By design, it looks for set environment variables: `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY`, or the default profile stored in your AWS credentials file (the same behavior as the AWS provider). There are also options to override the defaults.<sup>7</sup>

Next, we need a resource to test the S3 backend with. This can be any resource, but I personally like to use a special resource offered by the `null` provider called `null_resource`. There's lots of cool hacks you can do with `null_resource` and `local-exec` provisioners (which I'll delve into in the next chapter), but for now, all you need to know is that the following code provisions a dummy resource which prints "gotta catch em all" to the terminal during a `terraform apply`.

<sup>7</sup> <https://www.terraform.io/docs/backends/types/s3.html#configuration-variables>

**Listing 6.10 test.tf**

```
terraform {
  backend "s3" {
    bucket      = "pokemon-q56ylfpq6bzrw3dl-state-bucket"
    key         = "pikachu/thunderbolt"
    region     = "us-west-2"
    encrypt     = true
    profile     = "swinkler"
    role_arn   = "arn:aws:iam::215974853022:role/pokemon-q56ylfpq6bzrw3dl-tf-assume-role"
    dynamodb_table = "pokemon-q56ylfpq6bzrw3dl-state-lock"
  }
  required_version = "~> 0.12"
  required_providers {
    null = "~> 2.1"
  }
}

resource "null_resource" "motto" {
  triggers = {
    always = timestamp()
  }
  provisioner "local-exec" {
    command = "echo gotta catch em all" #A
  }
}
```

#A This is where the magic happens

Run a `terraform init`. The output is going to be a little different than what we've seen before, because now it's connecting to the S3 backend as part of the initialization process.

```
$ terraform init

Initializing the backend...

Successfully configured the backend "s3"! Terraform will automatically
use this backend unless the backend configuration changes.

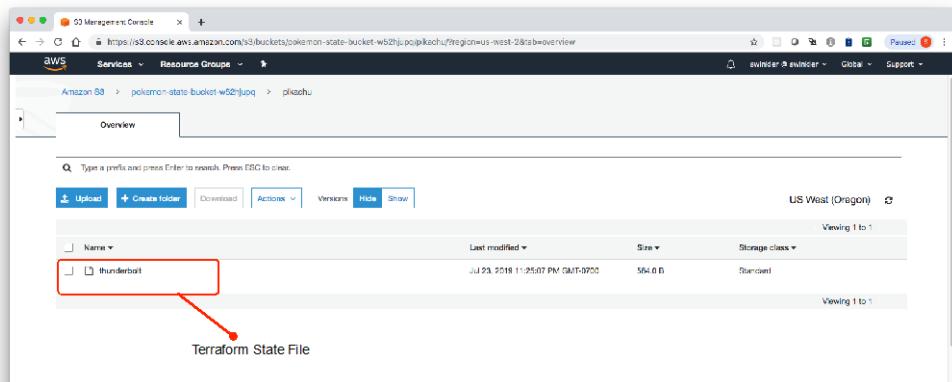
...
```

When it's finished initializing, run a `terraform apply -auto-approve`

```
$ terraform apply -auto-approve
null_resource.motto: Creating...
null_resource.motto: Provisioning with 'local-exec'...
null_resource.motto (local-exec): Executing: ["'/bin/sh" "-c" "echo gotta catch em all"]
null_resource.motto (local-exec): gotta catch em all #A
null_resource.motto: Creation complete after 0s [id=1806217872068888379] Apply complete! Resources: 1 added, 0
changed, 0 destroyed.
```

#A “gotta catch em all” is printed to stdout

As you can see, the `null_resource` output the catchphrase “gotta catch em all” to the terminal. Also, your state file is now safely stored in the S3 bucket created earlier, under the key “`pikachu/thunderbolt`”



**Figure 6.11** the state file is safely stored in the S3 bucket with key “`pikachu/thunderbolt`”

You can download the state file to view the contents, or manually upload a new version, although under normal circumstances there is no reason to do this. It’s much easier to view or manipulate the state file by running one of the `terraform state` commands, e.g:

```
$ terraform state list
null_resource.motto
```

### What Happens when Two People Apply at the Same Time?

In the unlikely event that two people are trying to deploy against the same remote backend at the same time, only one user will be able to acquire the state lock – the other will fail. The error message received will be:

```
$ terraform apply -auto-approve
Acquiring state lock. This may take a few moments...
```

```
Error: Error locking state: Error acquiring the state lock: ConditionalCheckFailedException: The conditional request failed
      status code: 400, request id: PNQMMJD6CTVVTSUPM537289FFVV4KQNSO5AEMVJF66Q9ASUAAJG
```

Lock Info:

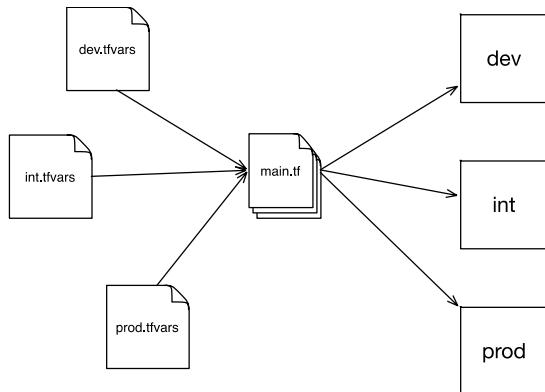
```
ID:      a494a870-6cad-f839-8a6b-9ac288eae7e4
Path:    pokemon-q56ylfpq6bzrw3dl-state-bucket/pikachu/thunderbolt
Operation: OperationTypeApply
Who:    swinkler@OSXSWINKMBP15.local
Version: 0.12.9
Created: 2019-11-25 02:47:45.509824 +0000 UTC
Info:
```

Terraform acquires a state lock to protect the state from being written by multiple users at the same time. Please resolve the issue above and try again. For most commands, you can disable locking with the "-lock=false" flag, but this is not recommended.

After the lock is released, the error message goes away and subsequent applies will succeed.

## 6.5 Reusing Configuration Code with Workspaces

Workspaces allow you to have more than one state file for the same configuration code. This means that you can deploy multiple environments without resorting to copy-pasting your configuration code into different folders. Each workspace can use its own variables definitions file to parameterize the environment (see figure 6.12).



**Figure 6.12** workspaces let you use the same configuration code, parameterized by different variables definitions files, to deploy to multiple environments

You have already been using workspaces without even realizing it. Whenever you perform a `terraform init`, Terraform creates and switches to a workspace named "default". You can prove this by running the command: `terraform workspace list`, which lists all workspaces and puts an asterisk next to the one you are currently on.

```
$ terraform workspace list
* default
```

To create and switch to a new workspace other than default, use the command: `terraform workspace select <workspace>`.

Why is this useful and why do you care? You could have just saved your state files under different names, such as "dev.tfstate" and "prod.tfstate" and point to them with a command like: `terraform apply -state=<path>`. Technically workspaces are no different than simply renaming state files. The reason you would use workspaces is because remote state backends

support workspaces and not the `-state` argument. This makes sense when you remember that remote state backends do not store state locally (so there is no state file to point to). I would recommend using workspaces even when using a local backend, if only to get in the habit of using them.

### 6.5.1 Deploying Multiple Environments with Workspaces

Our Pikachu deployment from earlier was a cute way to test that we could initialize and deploy against the remote stack backend, but it's impractical for describing how to use workspaces effectively. In this section we are going to try something more real-world. We will use workspaces to deploy two separate environments: "dev" and "prod". Each environment will be parameterized by its own variables' definition file, to allow us to customize each environment, for example to deploy to different AWS regions or accounts.

Create a new folder with a `main.tf` file as shown in Listing 6.11 (replace bucket, profile, role\_arn and dynamodb\_table as before).

#### **Listing 6.11 main.tf**

```
terraform {
  backend "s3" {
    bucket      = "<bucket>"
    key         = "team1/my-cool-project"
    region      = "<region>" #A
    encrypt     = true
    profile     = "<profile>"
    role_arn    = "<role_arn>"
    dynamodb_table = "<dynamodb_table>"
  }
  required_version = "~> 0.12"
  required_providers {
    null = "~> 2.1"
  }
}

variable "region" {
  description = "AWS Region"
  type       = string
}

provider "aws" {
  profile = "<profile>"
  region  = var.region
}

data "aws_ami" "ubuntu" {
  most_recent = true
  filter {
    name  = "name"
    values = ["ubuntu/images/hvm-ssd/ubuntu-bionic-18.04-amd64-server-*"]
  }
  owners = ["099720109477"]
```

```

}

resource "aws_instance" "instance" {
  ami      = data.aws_ami.ubuntu.id
  instance_type = "t2.micro"
  tags = {
    Name = terraform.workspace #B
  }
}

```

#A This region is the region in which your remote state backend lives and may be different than the region than you are deploying to. Since it is evaluated during initialization, it cannot be configured via a variable.  
#B this is a special variable, like "path", containing only one attribute: "workspace"

Within the current directory, create a folder called “variables” and create two files: `dev.tfvars` and `prod.tfvars`. The contents of the files will set the AWS region that the EC2 instance will be deployed to. An example of the variables definition file for `dev.tfvars` is shown in Listing 6.13.

### **Listing 6.13 dev.tfvars**

```
region = "us-west-2"
```

Next, we need to initialize the workspace, as per usual.

```
$ terraform init
...
Terraform has been successfully initialized!
```

You may now begin working with Terraform. Try running “terraform plan” to see any changes that are required for your infrastructure. All Terraform commands should now work.

If you ever set or change modules or backend configuration for Terraform, rerun this command to reinitialize your working directory. If you forget, other commands will detect it and remind you to do so if necessary.

Instead of staying on the “default” workspace, I suggest switching to a more appropriately named workspace right away. Most people name workspaces after a particular GitHub feature branch, or a deployment environment (such as dev, int, prod, etc.). Let’s switch to a workspace called “dev” to deploy the dev environment.

```
$ terraform workspace new dev
Created and switched to workspace "dev"!
```

You’re now on a new, empty workspace. Workspaces isolate their state, so if you run “terraform plan” Terraform will not see any existing state for this configuration.

Deploy the configuration code for the dev environment with the dev variables:

```
$ terraform apply -var-file=./variables/dev.tfvars -auto-approve
data.aws_ami.ubuntu: Refreshing state...
aws_instance.instance: Creating...
```

```
aws_instance.instance: Still creating... [10s elapsed]
aws_instance.instance: Still creating... [20s elapsed]
aws_instance.instance: Still creating... [30s elapsed]
aws_instance.instance: Creation complete after 38s [id=i-0b7e117464ae7eaa3]
```

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

The state file has now been created in the S3 bucket under the key: `env:/dev/team1/my-cool-project`. Let's now switch to a new prod workspace to deploy the production environment.

```
$ terraform workspace new prod
Created and switched to workspace "prod"!
```

```
You're now on a new, empty workspace. Workspaces isolate their state,
so if you run "terraform plan" Terraform will not see any existing state
for this configuration.
```

As we are in the new workspace, the state file is now empty, which we can verify by running a `terraform state list` command and noting that it returns nothing:

```
$ terraform state list
```

Deploying to the prod environment is similar to dev, except now we use `prod.tfvars` instead of `dev.tfvars`. I suggest specifying a different region for `prod.tfvars`, as shown in Listing 6.14.

#### **Listing 6.14 prod.tfvars**

```
region = "us-east-1"
```

Deploy to the prod workspace with the `prod.tfvars` variables definition file:

```
$ terraform apply -var-file=./variables/prod.tfvars -auto-approve
data.aws_ami.ubuntu: Refreshing state...
aws_instance.instance: Creating...
aws_instance.instance: Still creating... [10s elapsed]
aws_instance.instance: Still creating... [20s elapsed]
aws_instance.instance: Still creating... [30s elapsed]
aws_instance.instance: Creation complete after 38s [id=i-042808b20164b509d]
```

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

**NOTE** Since we are still using the same configuration code, you do not need to run `terraform init` again

Now in S3 we will have two state files, one for dev and one for prod (see figure 6.13).

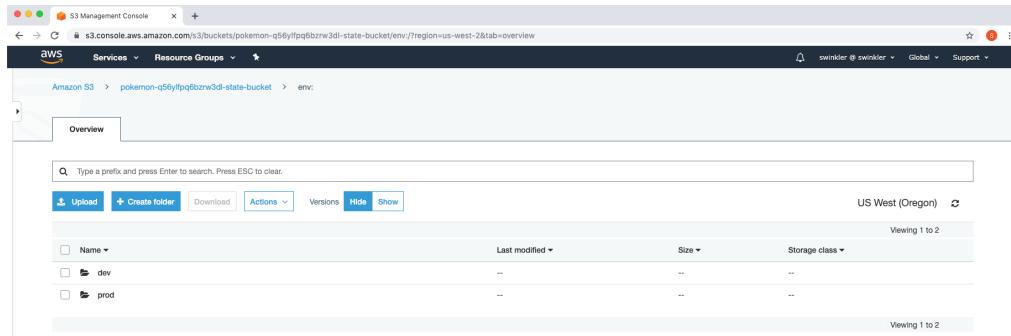


Figure 6.13 there are now two state files under :env, corresponding to the “dev” and “prod” workspaces

You can also inspect the two EC2 instances that were created, each named with the `terraform.workspace` variable (“dev” and “prod”, respectively).

**NOTE** I deployed both instances to the same region, rather than two different regions, so they would show up in the same screenshot

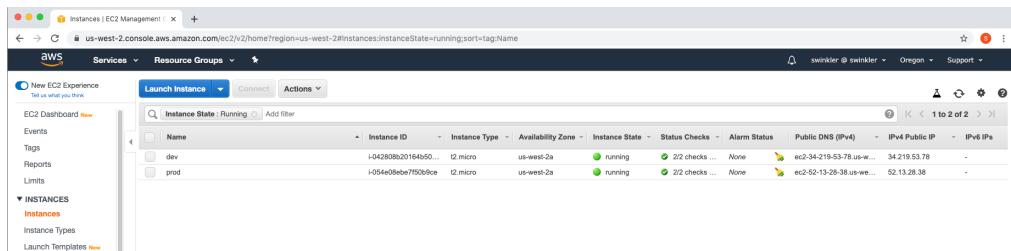


Figure 6.14 workspaces manage their own state files and their own resources. Here we can see two EC2 instances, one deployed from the dev workspace and one deployed from the prod workspace.

### 6.5.2 Clean up

We need to do two things: delete the EC2 instances from each environment and delete the S3 backend. You could directly delete the EC2 instances in the console, as that is the easiest thing to do, but it’s lazy and promotes bad habits. Instead, let’s do things the right way. First, delete the prod deployment:

```
$ terraform destroy -var-file=variables/prod.tfvars -auto-approve
data.aws_ami.ubuntu: Refreshing state...
aws_instance.instance: Refreshing state... [id=i-054e08ebe7f50b9ce]
aws_instance.instance: Destroying... [id=i-054e08ebe7f50b9ce]
aws_instance.instance: Still destroying... [id=i-054e08ebe7f50b9ce, 10s elapsed]
aws_instance.instance: Still destroying... [id=i-054e08ebe7f50b9ce, 20s elapsed]
aws_instance.instance: Still destroying... [id=i-054e08ebe7f50b9ce, 30s elapsed]
aws_instance.instance: Destruction complete after 32s
```

```
Destroy complete! Resources: 1 destroyed.
Releasing state lock. This may take a few moments...
```

Next, switch into the dev workspace and destroy that deployment:

```
$ terraform workspace select dev
Switched to workspace "dev".

$ terraform destroy -var-file=variables/dev.tfvars -auto-approve
data.aws_ami.ubuntu: Refreshing state...
aws_instance.instance: Refreshing state... [id=i-042808b20164b509d]
aws_instance.instance: Destroying... [id=i-042808b20164b509d]
aws_instance.instance: Still destroying... [id=i-042808b20164b509d, 10s elapsed]
aws_instance.instance: Still destroying... [id=i-042808b20164b509d, 20s elapsed]
aws_instance.instance: Still destroying... [id=i-042808b20164b509d, 30s elapsed]
aws_instance.instance: Destruction complete after 30s
```

```
Destroy complete! Resources: 1 destroyed.
Releasing state lock. This may take a few moments...
```

Finally, run a `terraform destroy` to clean up the provisioned S3 backend. You could destroy the state files created by the Pikachu dummy code, but it's not necessary as no "real" resources were created (another reason I like using the `null_resource`). Take down the S3 backend by switching back to the directory it was deployed from and `terraform destroy`.

```
$ terraform destroy -auto-approve
...
module.s3backend.aws_kms_key.kms_key: Still destroying... [id=16c6c452-2e74-41d4-ae57-067f3b4b8acd, 10s elapsed]
module.s3backend.aws_kms_key.kms_key: Still destroying... [id=16c6c452-2e74-41d4-ae57-067f3b4b8acd, 20s elapsed]
module.s3backend.aws_kms_key.kms_key: Destruction complete after 24s
```

```
Destroy complete! Resources: 8 destroyed.
```

**NOTE** Advanced state management techniques with the `terraform state` command are covered in chapter 9! This is helpful in the case of refactoring to prevent unnecessary destruction and re-creation of resources.

## 6.6 Introducing Terraform Cloud

Terraform Cloud is a HashiCorp SaaS product designed for using Terraform in multi-tenant environments. It brings collaboration features to Terraform, such as team management, Role-based access control (RBAC), and remote state storage. Terraform plans, applies and destroys are executed in a consistent and reliable containerized environment. Each workspace accesses its own shared state file and secret data, with logs and state file versions being archived for future reference. Additionally, you have the ability to create manual approval gates, send notifications through webhooks, govern the contents of Terraform configurations with Sentinel policies, and more.

Terraform Cloud is a solution marketed toward small to medium sized businesses and has two tiers of services offerings: a business tier and a free tier. The free tier is essentially a stripped-down, spartan version of their cushier service offerings, but it still gives you free remote state storage and team management.

Officially, the free tier of Terraform Cloud is touted as a form of charity to the community, by helping customers who are having trouble deploying and managing their own remote state backend. More likely, it was designed as a gateway toward getting you to buy one of their premium service offerings, such a higher tier of Terraform Cloud, or Terraform Enterprise (see figure 6.14).

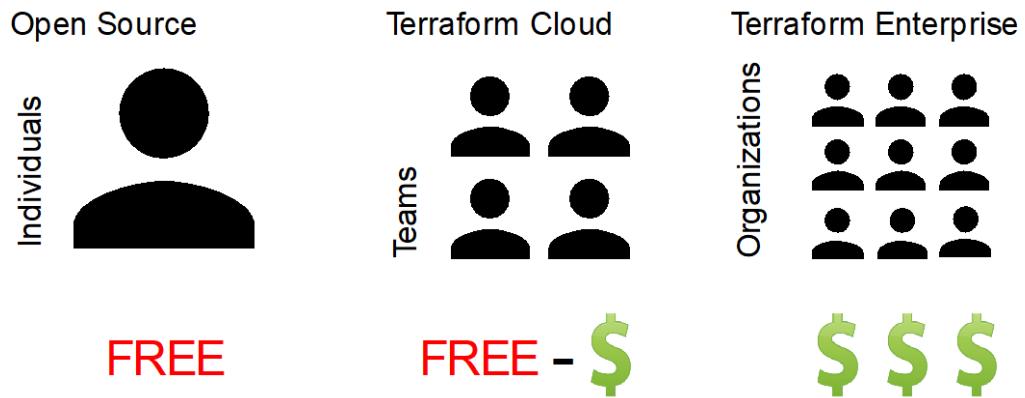


Figure 6.15 HashiCorp has three service offerings for Terraform: Terraform Open Source, Terraform Cloud, and Terraform Enterprise. Terraform Cloud comes in two tiers: free and business<sup>8</sup>

The remote state backend you get from Terraform Cloud does all the same thing that an S3 remote backend does: it stores state, locks and versions state files, encrypts state files at rest, and allows for fine-grained access control policies. But it also has a full workflow that lets you plan and apply remotely using the Terraform CLI, HTTP API or web UI, and lets you store secrets remotely so local environments never have access to them.

<sup>8</sup> <https://hashicorp.com/products/terraform/offering>

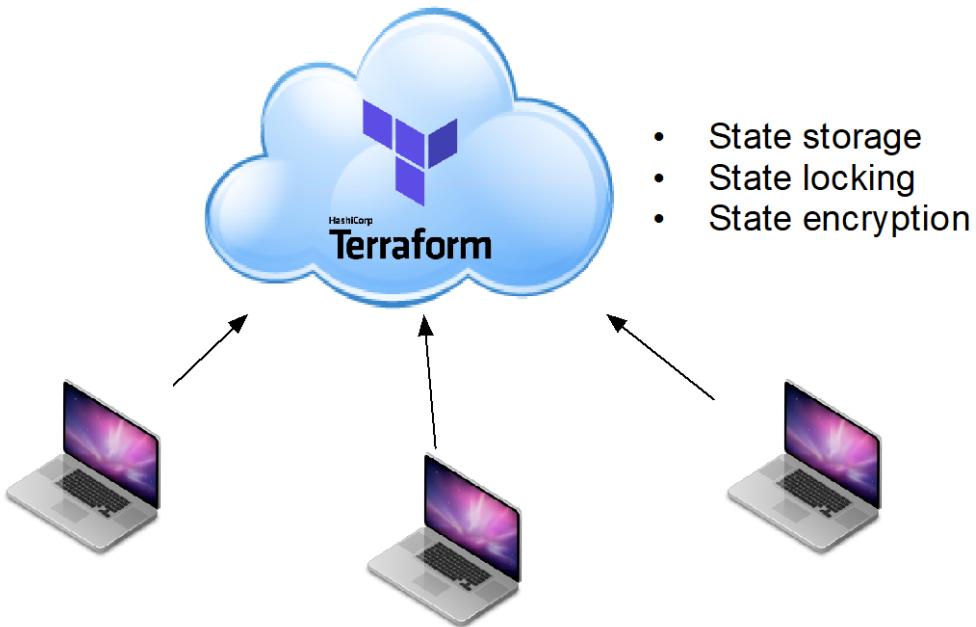


Figure 6.16 The free tier of Terraform Cloud is essentially a managed remote state backend with some convenient automation around it.

### 6.6.1 Getting Started with Terraform Cloud (Optional)

Deciding whether to use the free version of Terraform Cloud or not is entirely up to you, and whether you trust HashiCorp to store your data. I would recommend using the free version of Terraform Cloud if and only if you don't want to go through the trouble of managing your own remote backend or are already planning to migrate to a higher tier of Terraform Cloud/Terraform Enterprise at some point in the future. Otherwise, it makes more sense to stick with an S3 backend, or one of the other standard backends and develop your own CI/CD integrations around it. If you're still interested, or merely curious, then read on.

There are four steps to getting started with Terraform Cloud<sup>9</sup>. These are:

Making an account using your email address

Creating an organization for your workspaces

Obtaining a token for authenticating to the backend

<sup>9</sup> <https://app.terraform.io/signup>

## Deploying against the remote backend

It's pretty easy to do this yourself, but I've included some screenshots of the process to give you a feel for what to expect.

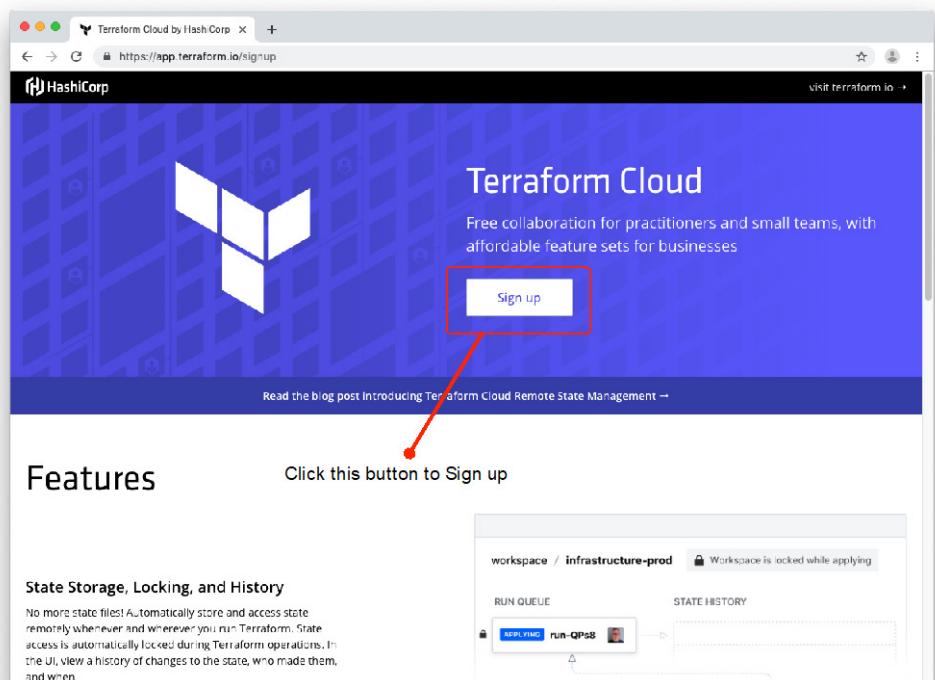


Figure 6.17 Sign-in page for Terraform Cloud. Click the big “Sign up” button to create an account

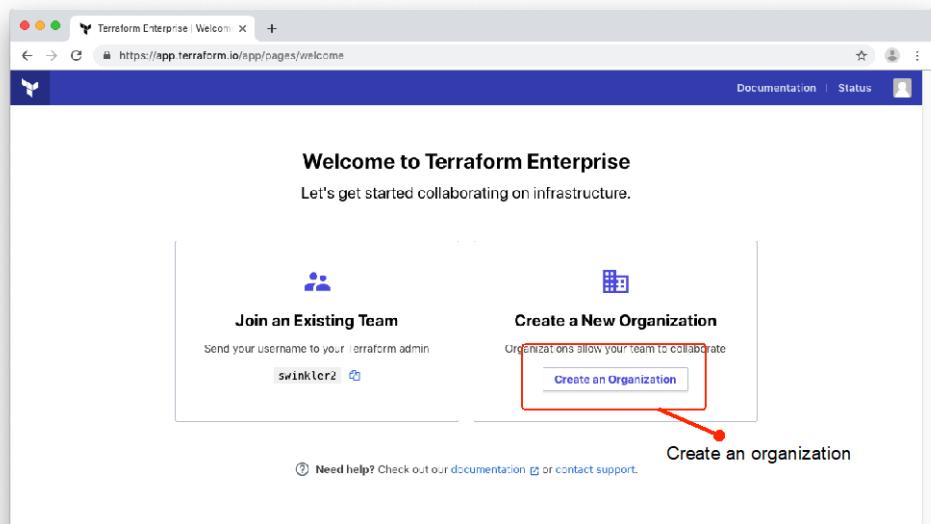


Figure 6.18 After logging in for the first time, you'll be prompted to create an organization with a name and an email address for the owner of the organization

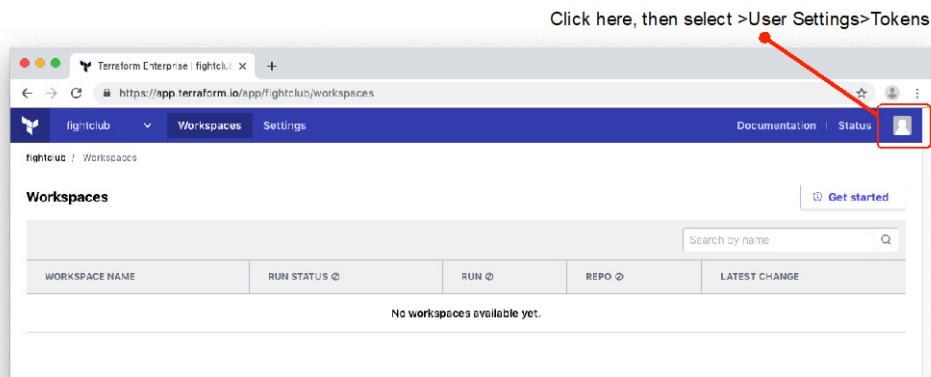


Figure 6.19 Once you've created an organization, select your avatar and navigate to the tokens tab

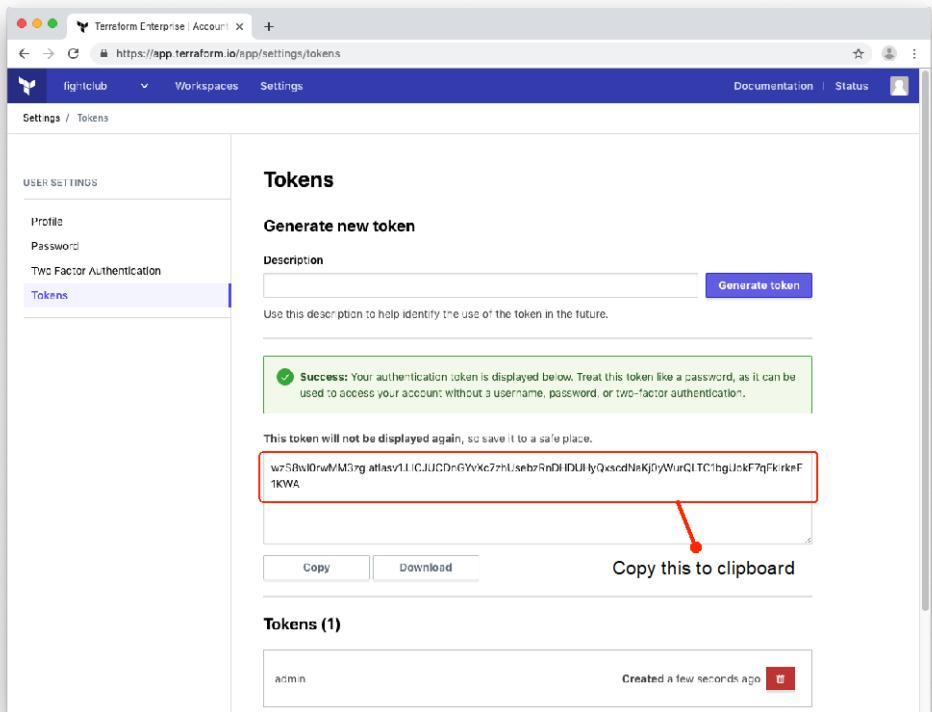


Figure 6.20 Copy the token into your clipboard. You'll need it for configuring the backend.

Once you have the token, you can now start using Terraform Cloud as a remote backend. An example backend configuration is shown in Snippet 6.2.

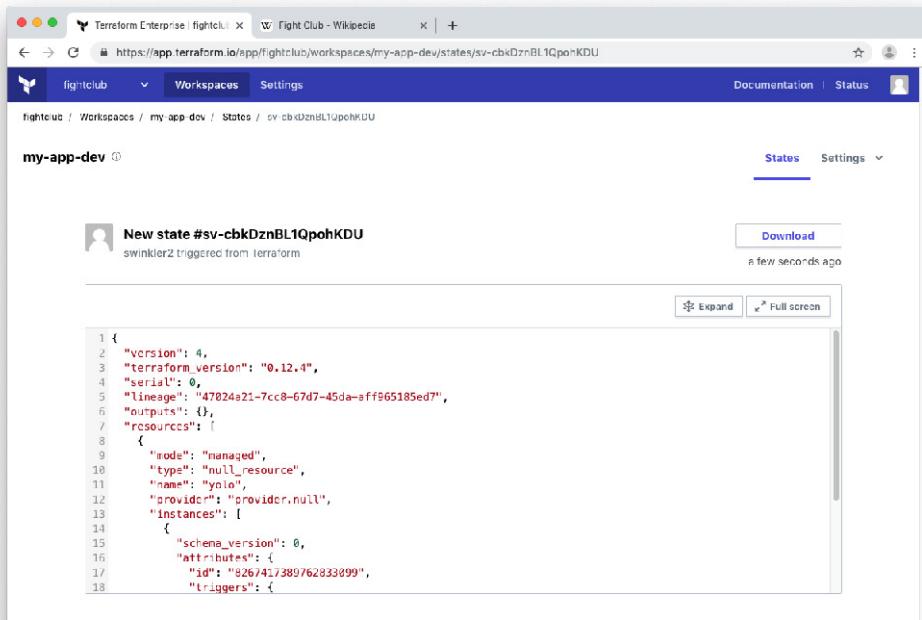
**NOTE** these tokens are not rotated. You would need to develop your own token rotation system using the API, and/or obtain keys from an secrets management system like HashiCorp's Vault

### Snippet 6.2 example backend configuration

```
terraform {
  backend "remote" {
    hostname = "app.terraform.io"
    organization = "fightclub"
    token = "hT7lx0C3wUOIIQ.atlasv1.OHfcX4Q11kSMDUTgbfdV2uws7AGYjm0xmBB8eyXbQWptyxYszlhP530FmSf1h4NyEfc" #A
    workspaces {
      name = "my-app-dev"
    }
  }
}
```

#A your token goes here

After applying, your workspace will automatically show up in the workspaces view. If you click on it, you can view and download a complete history of the state file for your workspace (see figure 6.20).



The screenshot shows a web browser window for Terraform Enterprise. The URL is <https://app.terraform.io/app/flightclub/workspaces/my-app-dev/states/sv-cbkDznBL1Qpo+hKDU>. The page title is "Fight Club - Wikipedia". The main navigation bar includes "fightclub", "Workspaces", "Settings", "Documentation", "Status", and a user icon. Below the navigation, the path "flightclub / Workspaces / my app dev / States / sv-cbkDznBL1Qpo+hKDU" is shown. A sub-navigation bar for "my-app-dev" has "States" selected. On the left, there's a sidebar with a profile picture and the text "New state #sv-cbkDznBL1Qpo+hKDU" followed by "swinkler2 triggered from terraform". On the right, there's a "Download" button and a timestamp "a few seconds ago". Below these are "Expand" and "Full screen" buttons. The main content area displays a JSON-formatted state file with line numbers from 1 to 18. The JSON content includes fields like "version", "terrafrom\_version", "serial", "lineage", "outputs", "resources", and "instances". One instance is highlighted with red annotations: "id: "267417389762333099", triggers": {".

```

1 {
2   "version": 4,
3   "terrafrom_version": "0.12.4",
4   "serial": 0,
5   "lineage": "47024a21-7cc8-67d7-45da-aff065185ed7",
6   "outputs": {},
7   "resources": [
8     {
9       "mode": "managed",
10      "type": "null_resource",
11      "name": "ynio",
12      "provider": "provider.null",
13      "instances": [
14        {
15          "schema_version": 0,
16          "attributes": {
17            "id": "267417389762333099",
18            "triggers": {

```

Figure 6.21 you can view and a download a complete history of the state file for your workspace

## 6.7 Fireside Chat

We've covered a lot of new information in this chapter. First, we started by talking about backends, and how it's important to use a remote backend for collaboration purposes. Then we designed an S3 backend module using a flat module structure, published it on the Terraform Module registry, and deployed it with a root module harness.

Next, we took a hard look at workspaces and how they behave within the framework of a remote state backend. Workspaces are technically equivalent to renaming state files, but they can be useful when reusing configuration code for multiple environments. Normally, each workspace will have its own variables definitions files, to allow setting variables and configuring providers uniquely for each workspace.

Finally, we examined using the remote backend offered by the free tier of Terraform Cloud. Terraform Cloud is one of HashiCorp's three service offerings, and a brief comparison between the three is outlined in Table 6.1.

**Table 6.1 Reference comparison for HashiCorp's three service offerings tiers for Terraform**

Name	Cost	Target	Description
Terraform Open Source	Free	Individuals and misers	Infrastructure as code provisioning and management
Terraform Cloud	Free - \$	Small to medium sized businesses	Collaboration features for practitioners and small teams
Terraform Enterprise	\$\$\$	Large organizations and enterprises	All open source capabilities plus collaboration and governance features for using Terraform across an organization

**NOTE** I did not mention anything about testing configuration code in this chapter, which is crucial for large scale infrastructure deployments and working on teams. I recommend looking into `terratest`<sup>10</sup> if you are interested in this topic.

## 6.8 Summary

- An S3 backend is used for remotely storing state files. It's made up of four components: 1) DynamoDB table 2) S3 bucket and KMS key 3) IAM Assume Role 4) Housekeeping resources
- Flat Module structures are when you organize code by file, rather than by module. For some situations, it can be more concise compared to Nested Modules, but can also lead to spaghetti code.
- Modules may be shared through a variety of means including, but not limited to, S3 buckets, GitHub repos, and the Terraform Module Registry
- Workspaces are a way to reuse the same configuration code for deploying to multiple environments. There are built-in CLI commands that make this easy to do.
- Terraform Cloud and Terraform Enterprise are for teams and large organizations respectively. Terraform Cloud has a free tier that gives you free remote state management, which can save you the time and trouble of deploying your own backend.

<sup>10</sup> <https://www.github.com/gruntwork-io/terratest>

# 7

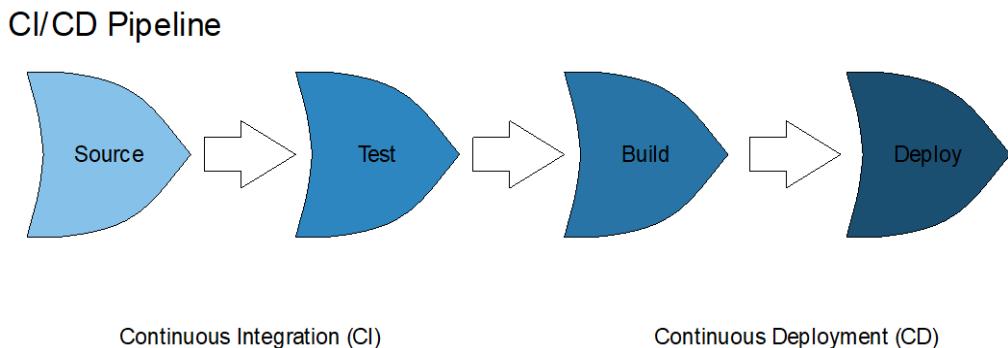
## *CI/CD Pipelines as Code*

### **This chapter covers:**

- Designing an end-to-end CI/CD pipeline for Docker containers on GCP
- Enacting a two-stage strategy for deploying static and dynamic infrastructure
- Iterating over complex types with `for-each` and dynamic blocks
- Explicitly overriding default providers at the resource and module level
- Creating custom resources with `local-exec` provisioners

CI/CD stands for *Continuous Integration (CI)* and *Continuous Deployment (CD)*. It's a DevOps methodology for enabling development teams to quickly ship code changes to production environments. CI/CD means changes are integrated faster, which results in smaller changes and decreased risk. This, in combination with the agility to deliver features faster is the key successes of CI/CD.

CI/CD is typically implemented by a CI/CD *pipeline*, which describes how code gets from version control systems through to end users and customers. In a CI/CD pipeline, each stage performs some discreet task, such as building, testing, and deploying, before automatically promoting to the next stage in the pipeline (see figure 7.1).



**Figure 7.1** CI/CD stands for Continuous Integration (CI) and Continuous Deployment (CD). It's normally implemented by a CI/CD pipeline, in which one stage automatically promotes into the next

In this chapter, we'll be using Terraform to provision a CI/CD pipeline for *Docker containers*<sup>1</sup> on Google Cloud Platform (GCP). GCP is the third smallest of the four major clouds (AWS, Azure, GCP and AliCloud), but they've seen a lot of growth recently, and for good reason: they have a clean, well thought out UI and offer many appealing cloud services. Notably, GCP is the industry leader for container orchestration and is the brains behind Kubernetes (a popular container orchestration platform). The only thing I do not like about GCP is the implementation of their Terraform provider, which can sometimes be awkward to use. We will talk about some of these pain points in this chapter, as well as various workarounds.

We'll start by covering the last few syntax and expression elements that we haven't had a chance to introduce earlier. Specifically, we'll introduce *for-each expressions*, *dynamic blocks*, and *resource provisioners*. Although we've seen dynamic and functional programming in chapter 2, these new configuration constructs enable writing much more powerful, expressive, and dynamic code than ever before.

Resource provisioners are remarkable because they can call external scripts through a backdoor of the Terraform runtime. We will see how to use `local-exec` resource provisioners attached to a `null_resource` to create custom resources without having to go through the effort of writing a Terraform provider. This isn't best practice, but it is good to be aware of, and for some edge cases (like unsupported or buggy resources) it may be your only option.

Once our CI/CD pipeline is up and running, we'll verify that it's functioning properly by watching it build and deploy a Docker container after being triggered by a commit to a Git source

<sup>1</sup> Docker containers are lightweight, standalone, executable packages of software that include everything needed to run an application: code, runtime, system tools, system libraries and settings

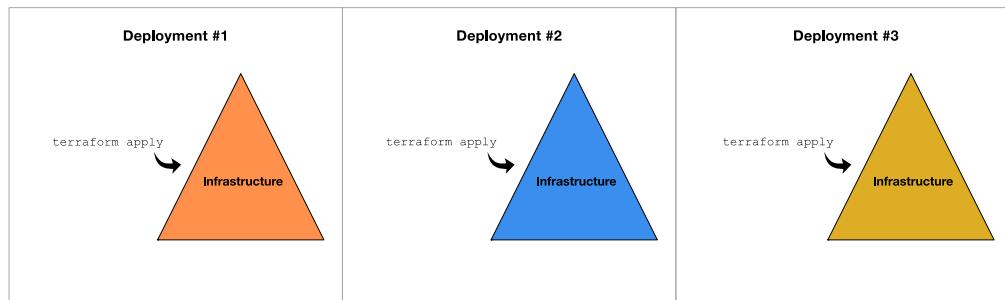
repository. The Docker container that is deployed will run a simple “Hello, World!” style web app.

Note that there are actually two deployments, the first being performed with Terraform, and the second with CI/CD. This two-stage strategy of deploying static infrastructure with Terraform, and dynamic infrastructure with some other tool, is a great technique that we will explore more in the next few chapters. Practically speaking, it allows for faster deployments and mitigates risk if something goes wrong. Moreover, this strategy provides an excellent way to integrate Terraform with other deployment technologies (such as Ansible, Chef, Puppet, etc.) to better suit your business requirements.

## 7.1 A Tale of Two Deployments

How we’ve previously deployed applications with Terraform is convenient (because everything can be deployed with a single `terraform apply`), but much slower than it could be. Modern software development methodologies, such as Agile, demand faster iteration cycles so that code changes can quickly make their way into production. This usually means being able to test, build, and deploy to production through some kind of automated CI/CD pipeline. As much as I love Terraform, and as well optimized as it is, generating execution plans are incredibly slow – doubly so if you have many resources in the state file. When all you want to do is update the code for a Docker container, and not touch any of the underlying infrastructure, Terraform can feel downright sluggish.

An easy optimization is to isolate the infrastructure that “changes a little” from the infrastructure that “change a lot”. I call these *static* and *dynamic* infrastructures, respectively. By first deploying static infrastructure, you have a solid platform on which to deploy dynamic infrastructure. Ideally, changes to the dynamic infrastructure should not compromise the integrity or immutability of the static layer. By architecting your project in such a way, you’re able to experience a much faster Software Development Lifecycle (SDLC) compared to deploying your infrastructure and application layers concurrently. This can be visualized by figures 7.2 and 7.3.



**Figure 7.2 Redeploying an entire stack each time you want to make a change is slow**

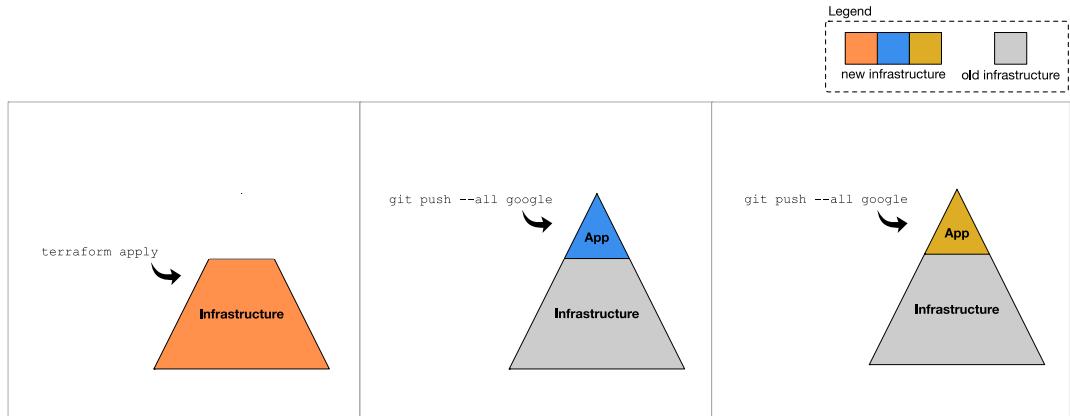


Figure 7.3 By separating your project into “what changes a lot” vs. “what changes a little” you can experience higher agility

**TIP** you could also use two Terraform deployments, one to deploy static infrastructure and one to deploy dynamic infrastructure. An example of this would be using one Terraform workspace to deploy a Kubernetes cluster, and another to deploy Helm charts.

## 7.2 CI/CD for Docker Containers on GCP

Docker containers are an excellent way to package your code and ensure that it has all the resources and libraries required to run, while still being portable across multiple environments. Because of the enormous popularity of containers, there exists many tools and established architecture patterns for setting up a CI/CD pipeline. We'll take advantage of some excellent managed GCP services to deploy a complete CI/CD pipeline for building, testing and deploying Docker images.

### 7.2.1 Designing the Pipeline

KNative is an abstraction layer over Kubernetes that allows for running and managing serverless workloads with ease. It is the backbone for a new GCP service called “Cloud Run”, which we'll use to handle horizontal scaling, load balancing, and DNS resolution for containers. The purpose of Cloud Run is primarily to simplify the scenario some, as it would be much more complex having to deploy and configure an entire Kubernetes cluster in addition to all the other topics we are going to cover in this chapter.

**NOTE** if you already have a Kubernetes cluster, this scenario could be modified to use that as a deployment target instead of Cloud Run. This is left as an exercise to the reader.

As mentioned earlier, CI/CD pipelines for containers generally involve stages for building, testing, publishing, and deploying code into production environments. Preferably, you would

have multiple environments (e.g. dev, int, prod), but for this scenario, we only have a single environment (prod). The focus is more on the CI, rather than the CD part of CI/CD.

In addition to Cloud Run, we'll be using the following managed GCP services for constructing the CI/CD pipeline:

- Cloud Source Repositories – a version-controlled Git source repository
- Cloud Build – For testing, building, publishing and deploying the container
- Container Registry – For storing the image build artifacts
- Cloud Run – For running serverless containers on a managed Kubernetes cluster

The steps of the pipeline that we'll be building is shown in figure 7.4.

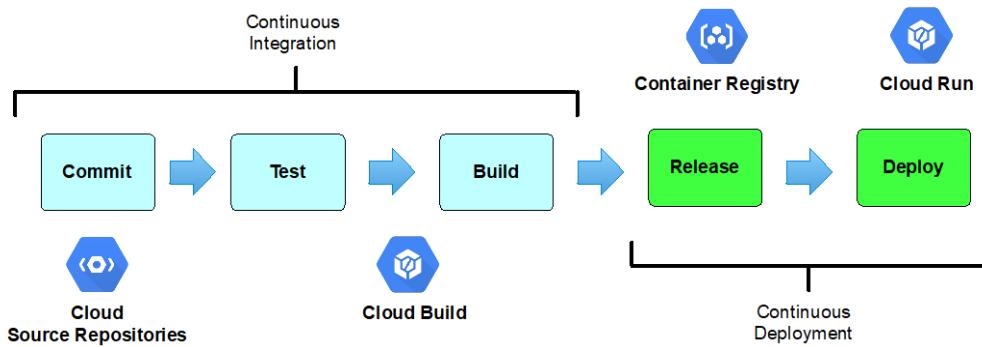


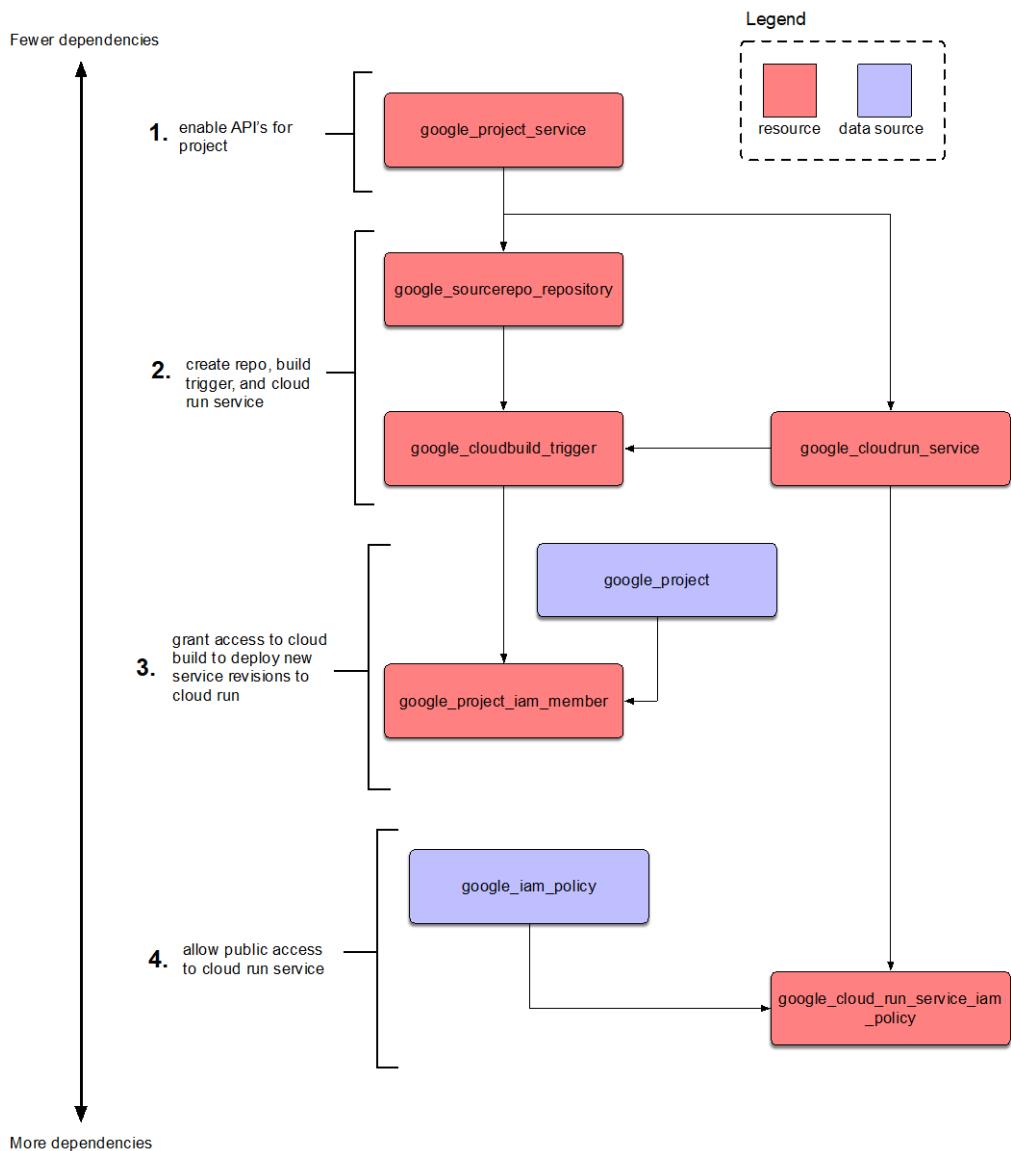
Figure 7.4 CI/CD pipeline for GCP. Commits to Cloud Source Repositories triggers a build in Cloud Build, which then publishes an image in the Container Registry and kicks off a new deployment to Cloud Run.

## 7.2.2 Detailed Engineering

This project doesn't have much in the way of code, but the code it does have is tricky. There are three main components that make up the code for the CI/CD pipeline, these being:

1. **Enabling API's** – GCP requires that you explicitly enable that APIs you wish to use
2. **CI/CD Pipeline** – provision and wire up the stages of the CI/CD pipeline
3. **Cloud Run Service** – run the serverless container on GCP

A dependency diagram is shown in figure 7.5.



**Figure 7.5** There are three sets of components: one for enabling the API's, one for provisioning the CI/CD pipeline, and one for configuring the Cloud Run service

## 7.3 Initial Workspace Setup

If you do not already have service account credentials for GCP, you will need to get them<sup>2</sup>. There are a few steps involved in the process that may not be obvious, especially if you are new to Google Cloud. The following sidebar has more information on what needs to be done to acquire service account credentials for GCP.

### Acquiring Service Account Credentials for GCP

GCP makes it challenging to get service account credentials for Terraform, at least compared to AWS or Azure.

Assuming you have a brand-new account with nothing in it, the procedure is as follows:

Create a project

Link billing account to project

Create service account

Grant IAM role to service account (usually Project Owner)

Create and download access key for service account

Enable base service APIs (for Cloud Resource Manager and Service Usage)

For your convenience, I've included a small script<sup>3</sup> to help in automating this task in the online source code for this book. Alternatively, you can authenticate using the Cloud SDK, but this is not recommended as not all API's are compatible with this method<sup>4</sup>.

### 7.3.1 Organizing the Directory Structure

There are two parts to this project, the part deployed with Terraform and the part not deployed with Terraform. Easy, right? Well how do you organize code that's related to a central project, but different enough that it should still be kept separate? Monorepos, of course! Normally, I organize my code by creating a single project directory with two subdirectories, one for all things Terraform related (i.e. static infrastructure), and another for the application code (i.e. dynamic infrastructure). Do this now by creating a project folder, for example "gcp-pipelines", with two subfolders, "infrastructure" and "application". When you're done with that, switch into the infrastructure folder, as this will be the working directory for the root module.

In the infrastructure folder, create a `variables.tf` file with the following content:

#### **Listing 7.1 variables.tf**

```
variable "project_id" {
  description = "The GCP project id"
  type = string
}
```

<sup>2</sup> <https://cloud.google.com/iam/docs/creating-managing-service>

<sup>3</sup> <https://www.github.com/scottwinkler/manning-code/blob/master/chapter7/complete/bootstrap.sh>

<sup>4</sup> [https://www.terraform.io/docs/providers/google/provider\\_reference.html#full-reference](https://www.terraform.io/docs/providers/google/provider_reference.html#full-reference)

```

variable "region" {
  default = "us-central1"
  description = "GCP region"
  type = string
}

variable "namespace" {
  description = "The project namespace to use for unique resource naming"
  type = string
}

```

Next, create a `terraform.tfvars` file. You can change the region and namespace if you like, but set `project_id` to the ID of your GCP project.

### **Listing 7.2 terraform.tfvars**

```

project_id=<your_project_id> #A
namespace = "team-pokemon"
region = "us-central1"

```

#A your GCP project id goes here

Notice that the namespace is “team-pokemon”. Imagine, if you will, that this pipeline wasn’t just any old pipeline, but was going to be used by a group of millennial developers to deploy their hip new Pokémon themed app. This represents the fact that A) the following Terraform code is reusable, and B) if you are an expert in CI/CD, you will always be asked to do work for other people. It’s a lot like being the only tech person in the family: you can guarantee people will come along asking for help.

Finally, we need to declare the “google” provider. Create a `providers.tf`, with contents shown below.

### **Listing 7.3 providers.tf**

```

provider "google" {
  credentials = file("account.json")
  project    = var.project_id
  region     = var.region
}

```

## **Implicit vs Explicit Providers**

Google Cloud Platform maintains two provider builds in the provider registry: a “google” provider and a “google-beta” provider. The beta provider implements resources that are not yet ready to be merged into the “google” provider. For example, until recently, the Cloud Run service was only available as part of the “google-beta” provider which meant that it was necessary to configure both GCP providers side by side so that Cloud Run resources could be provisioned by explicitly passing in the beta provider.

Passing a provider explicitly means that you override the implicit (or default) provider with another provider. This applies not only the “google-beta” provider, but for duplicate declarations of the same provider as well. When more than one

provider declaration is present, one provider will always be designated as the default provider, while any others are designated as non-default providers.

**TIP** passing providers explicitly is most commonly used for multi-region deployments. For instance, you could deploy resources to both us-central1 and us-west1 using two different configurations of the same provider.

Non-default providers can be explicitly passed into both resources and modules, making it possible to use multiple configurations of the same provider within a given workspace. Figure 7.6 illustrates how the “google-beta” provider can be used to override the implicit provider for the Cloud Run service resource, which was the only way to deploy this resource before it was added to the “google” provider.

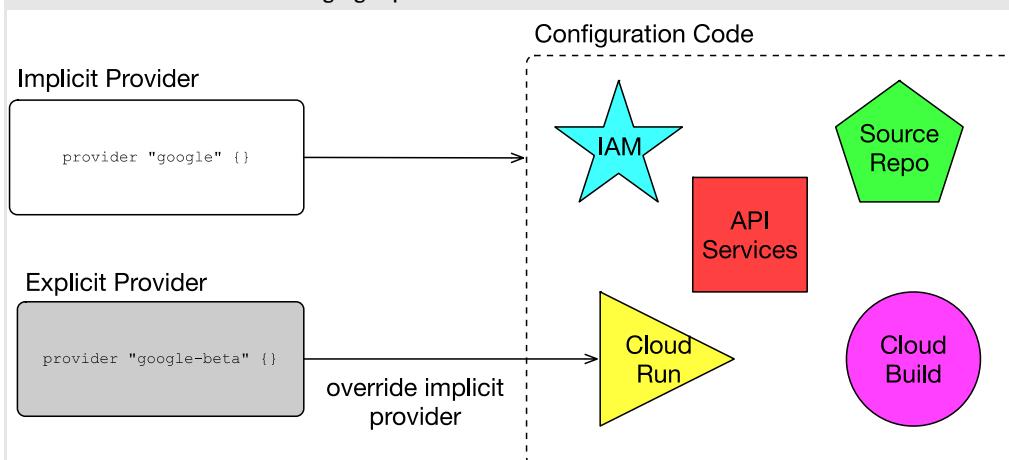


Figure 7.6 resources and modules have the option to override implicit providers with explicit providers. Beta services not supported by the “google” provider can be provisioned by explicitly passing in the “google-beta” provider.

## 7.4 Dynamic Configurations and Provisioners

Google is highly strict and opinionated when it comes to matters of IAM. For example, in a new project, you must purposefully enable the APIs of the services you wish to use before you can use them. This can be done in Terraform with the `google_project_service` resource, as long as this resource is created first. Below is the code for enabling the services that we will need in this project. There’re two new features here that you haven’t seen before, `for_each`, and `local-exec`. I’ll explain these in the next sections.

### Listing 7.4 main.tf

```

locals {
  services = [ #A
    "sourcerepo.googleapis.com",
    "cloudbuild.googleapis.com",
    "run.googleapis.com",
    "iam.googleapis.com",
  ]
}

```

```

}

resource "google_project_service" "enabled_service" {
  for_each = toset(local.services)
  project = var.project_id
  service = each.key

  provisioner "local-exec" { #B
    command = "sleep 60"
  }

  provisioner "local-exec" { #C
    when   = destroy
    command = "sleep 15"
  }
}

#A list of service APIs to enable
#B creation-time provisioner
#C destruction-time provisioner

```

#### 7.4.1 For-Each vs. Count

The `for_each` meta-attribute accepts a map, or set of strings, and creates an instance for each element in that map or set. Although analogous to loop constructs in other programming languages, `for_each` does **not** guarantee sequential iteration (because sets and maps are inherently unordered collections). For-each is most similar to the meta-attribute `count`, but has a number of distinct advantages, namely:

1. **Intuitive** –For-each is a much more natural concept, compared to iterating by index
  2. **Less verbose** – syntactically, `for_each` is shorter and more pleasing to the eye
  3. **Ease of use** – instead of storing instances in an array, instances are stored in a map.
- This makes individual resource instances much easier to reference.

For-each is the recommended approach for creating dynamic configurations, unless you have specific reason to access something by index (such as our round robin approach to creating Mad Lib files in chapter 3). The syntax of `for_each` is highlighted below in figure 7.7

```

resource "google_project_service" "enabled_service" {
  for_each = toset(local.services) →
  project  = var.project_id
  service   = each.key →
} →

```

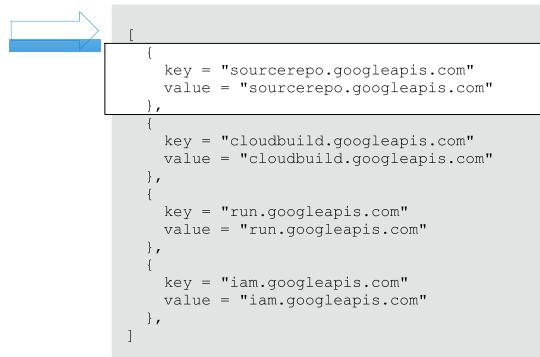
Map or set to iterate  
Current key accessor

Figure 7.7 syntax of the `for_each` meta-attribute and associated `each` object

In resource blocks where `for_each` is set, an additional `each` object is made available for use by expressions. The `each` object is a reference to the current entry in the iterator, and has two accessors:

1. `each.key` – the map key or set item corresponding to this entry
2. `each.value` – the map value corresponding to this entry (for sets, this is the same as `each.key`)

I personally found `each` to be confusing when I first read about it – after all, what do keys and values have to do with sets? What helped me was imagining that Terraform first transforms the set into a list of objects, then iterates over that list (see figure 7.8).



**Figure 7.8** the input set is transformed into a list of each objects. This new iterator is used by For-Each.

When `for_each` is set, the resource address points to a map of resource instances, rather than a single instance (or list of instances, as would be the case with `count`). To refer to a specific instance member, simply append the iterator map key after the normal resource address, i.e. `<TYPE>. <NAME>. [<KEY>]`. For example, if we wanted to refer to the resource instance corresponding to "sourcerepo.googleapis.com", we could do so with the following expression:

```
google_project_service.enabled_service["sourcerepo.googleapis.com"]
```

### 7.4.2 Executing Scripts with Provisioners

Resource provisioners allow you to execute scripts on local or remote machines as part of resource creation or destruction. They are normally used for resource bootstrapping and clean up, such as copying files onto an instance, starting up configuration management clients, hacking into the mainframe, etc. All resources support provisioner blocks, but most of the time you do not need them.

**NOTE** because resource provisioners call external scripts, there is an implicit dependency on the OS interpreter, for example to the bash shell.

Provisioners allow you to dynamically extend functionality on resources by hooking into resource lifecycle events. There are two kinds of resource provisioners:

1. Creation-Time Provisioners
2. Destruction-Time Provisioners

Most people who use provisioners exclusively use creation-time provisioners, for example to run a script or kick off some task. Snippet 7.1 is rather unusual because it uses both.

#### Snippet 7.1 local-exec provisioner

```
resource "google_project_service" "enabled_service" {
  for_each = toset(local.services)
  project = var.project_id
  service = each.key

  provisioner "local-exec" {
    command = "sleep 60" #A
  }

  provisioner "local-exec" {
    when = destroy
    command = "sleep 15"
  }
}
```

#A The “when” attribute defaults to “apply” if not set

The above creation-time provisioner invokes the command `sleep 60` to wait for 60 seconds *after* `Create()` has exited, but before the resource is marked as “created” by Terraform (see figure 7.9). Likewise, the destruction-time provisioner waits for 15 seconds *before* `Delete()` is called (see figure 7.10). Both of these pauses (determined experimentally through trial and error!) are essential to avoid potential race conditions when enabling/disabling service APIs.

---

#### TIMING IS EVERYTHING

Why are race conditions happening in the first place? Can't this be solved with a simple `depends_on`? In an ideal world, yes. Resources should always be completely created before they report themselves as created – that way there will be no race conditions occurring during resource provisioning. Unfortunately, the people who create Terraform providers (and the APIs that the providers leverage) are not perfect. Sometimes resources are marked as “created” when actually it takes a few more seconds before they are truly ready. By inserting delays with the local-exec provisioner, you can solve many of these strange race condition style bugs.

If you do encounter a race condition bug like this, you should always file an issue with the provider owner. Be forewarned however, that issues frequently take a long time to solve (possibly being contingent on upstream APIs being modified) and you cannot rely on someone else to solve your problem exactly when you need it. Adding in delays with local-exec provisioners is a stopgap measure to ensure that resources are always created at the right time, even if the

provider is wonky. It's a hack to allow you to work around the problem and should not be used under normal circumstances.

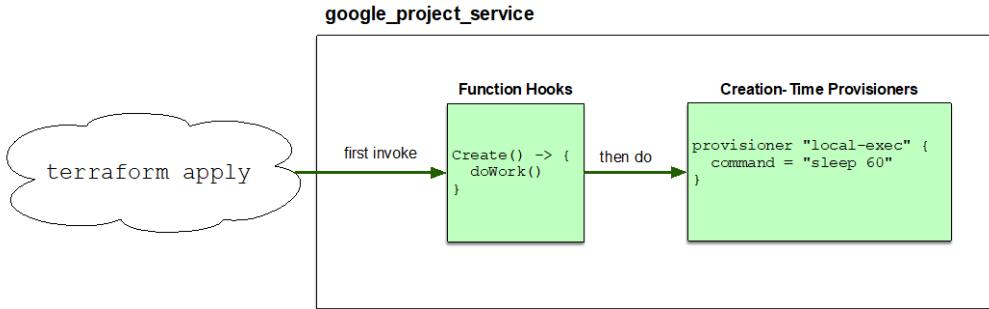


Figure 7.9 the local-exec provisioner is called after the Create() function hook has exited, but before the resource is marked as “created” by Terraform

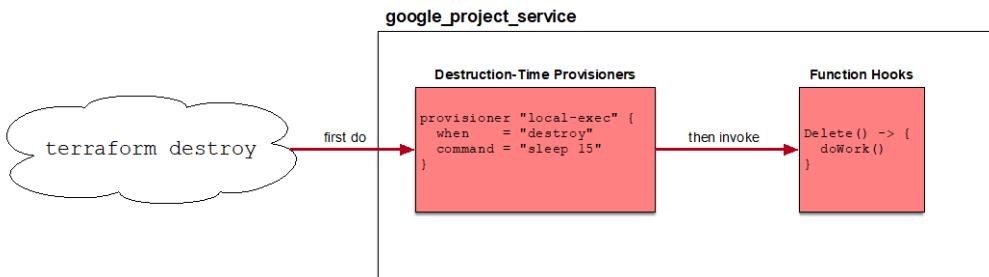


Figure 7.10 the local-exec provisioner is called before Delete()

#### 7.4.3 Null Resource with a Local-Exec Provisioner

If a creation-time and destruction-time provisioner are attached to a `null_resource`, you can create a sort of custom Terraform resource. A `null_resource` is part of the `null` provider and takes part in the same lifecycle as every other resource, but performs no actions of its own. A creation-time provisioner could be used to call a script to create something, and a destruction-time provisioner could call a script to delete it afterwards.

Snippet 7.3 shows sample code for creating a custom resource that prints “Hello World!” on resource creation and “Goodbye cruel world!” on resource deletion. I’ve spiced it up a bit by using `cowsay`, a CLI tool that generates ASCII pictures of a cow with a message.

##### **Snippet 7.2 custom cowsay resource**

```
resource "null_resource" "cowsay" {
```

```

provisioner "local-exec" { #A
  command = "cowsay Hello World!"
}

provisioner "local-exec" { #B
  when = destroy
  command = "cowsay -d Goodbye cruel world!"
}
}

```

#A creation-time provisioner  
#B destruction-time provisioner

On `terraform apply`, Terraform will run the creation-time provisioner:

```

$ terraform apply -auto-approve
null_resource.cowsay: Creating...
null_resource.cowsay: Provisioning with 'local-exec'...
null_resource.cowsay (local-exec): Executing: ["#!/bin/sh" "-c" "cowsay Hello world!"]
null_resource.cowsay (local-exec): _____
null_resource.cowsay (local-exec): < Hello World! >
null_resource.cowsay (local-exec): -----
null_resource.cowsay (local-exec):   \ ^__^
null_resource.cowsay (local-exec):   \ (oo)\_____
null_resource.cowsay (local-exec):   (__)\    )\/\
null_resource.cowsay (local-exec):   ||----w |
null_resource.cowsay (local-exec):   ||     |
null_resource.cowsay: Creation complete after 0s [id=1729885674162625250]

```

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

Likewise, on `terraform destroy`, Terraform triggers the destruction-time provisioner:

```

$ terraform destroy -auto-approve
null_resource.cowsay: Refreshing state... [id=1729885674162625250]
null_resource.cowsay: Destroying... [id=1729885674162625250]
null_resource.cowsay: Provisioning with 'local-exec'...
null_resource.cowsay (local-exec): Executing: ["#!/bin/sh" "-c" "cowsay -d Goodbye cruel world!"]
null_resource.cowsay (local-exec): _____
null_resource.cowsay (local-exec): < Goodbye cruel world! >
null_resource.cowsay (local-exec): -----
null_resource.cowsay (local-exec):   \ ^__^
null_resource.cowsay (local-exec):   \ (xx)\_____
null_resource.cowsay (local-exec):   (__)\    )\/\
null_resource.cowsay (local-exec):   U  ||----w |
null_resource.cowsay (local-exec):   ||     |
null_resource.cowsay: Destruction complete after 0s

```

Destroy complete! Resources: 1 destroyed.

### The Dark Road of Resource Provisioners

Although resource provisioners can be incredibly useful in a pinch, they should be avoided whenever possible. Resource provisioners undermine Terraform's ability to actually manage your resources for you and can get you stuck in a bad state if something goes wrong.

The main advantage of using Terraform is that it's declarative and therefore all code for provisioning is assumed to be version controlled and tested. When you make calls out to external ad hoc scripts, you challenge that baseline assumption. Beware that Terraform does not keep track of changes to provisioners in the same way it does for resources attributes; no copy is stored in the state file and there is no way to calculate diffs. If anything goes wrong, you are out of luck.

Since resource provisioners are not monitored in the same way that resource attributes are, if there is an error in the code, or something unexpected occurs there's no way to recover. Some of the worst Terraform bugs I have ever encountered have been the result of a module authors overreliance on resource provisioners. You can't destroy, you can't apply, you're just stuck. And it feels terrible. HashiCorp has stated that resource provisioners are an anti-pattern and they may even be deprecated entirely in a newer version of Terraform. I somehow doubt this, but it's good to be mindful of nonetheless.

**TIP** If you are interested in creating custom resources without writing your own provider, I recommend taking a look at the Shell provider<sup>5</sup>, which allows for better integration with the Terraform runtime than what is possible with null\_resources and local-exec provisioners

#### 7.4.4 Dealing with Repeating Configuration Blocks

Getting back to the main scenario, we need to declare the resources that make up the CI/CD pipeline. To start, add the following code from Listing 7.5 into your `main.tf`. This will provision a version-controlled source repository, the first stage in our CI/CD pipeline.

<sup>5</sup> <https://www.terraform.io/docs/providers/type/community-index.html>

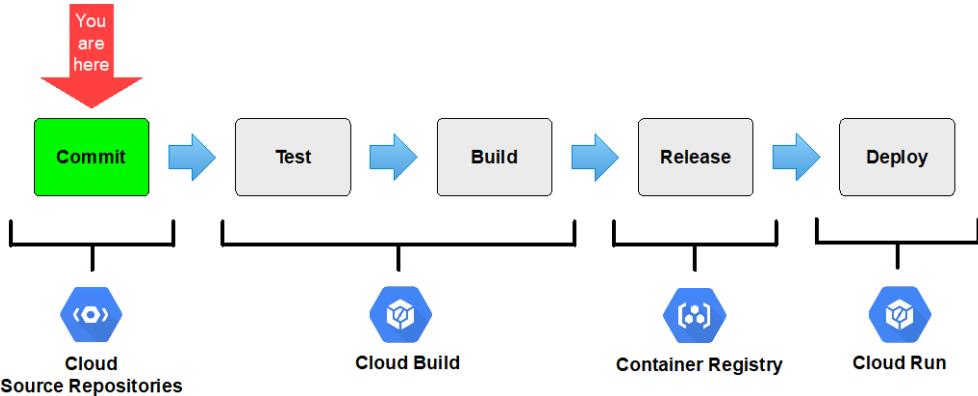


Figure 7.11 CI/CD pipeline - Stage 1 of 3

#### **Listing 7.5 main.tf**

```

resource "google_sourcerepo_repository" "repo" {
  depends_on = [
    google_project_service.enabled_service["sourcerepo.googleapis.com"]
  ]

  name      = "${var.namespace}-repo"
}
    
```

Next, we need to set up Cloud Build to trigger a build from a commit to the source repository. Since there are a number of steps in the build process, we could use a series of repeating configuration blocks, as shown in Snippet 7.2.

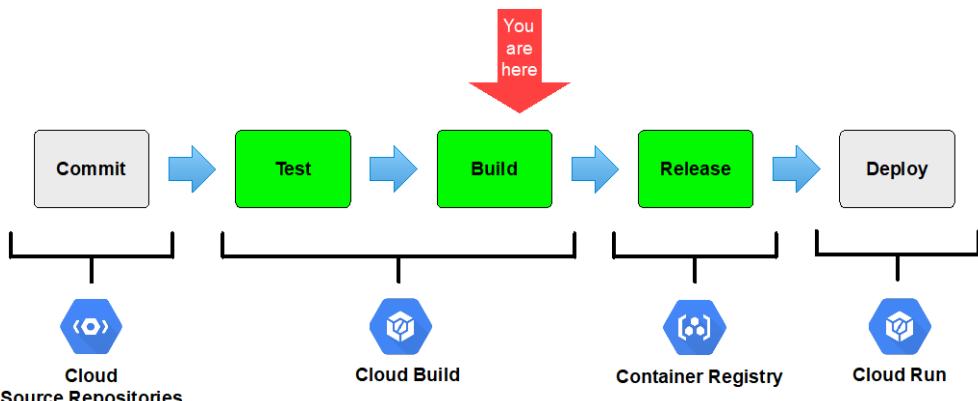


Figure 7.12 CI/CD pipeline - Stage 2 of 3

### Snippet 7.3 repeating configuration blocks

```
resource "google_cloudbuild_trigger" "trigger" {
  depends_on = [
    google_project_service.enabled_service["cloudbuild.googleapis.com"]
  ]

  trigger_template {
    branch_name = "master"
    repo_name  = google_sourcerepo_repository.repo.name
  }

  build {
    step { #A
      name = "gcr.io/cloud-builders/go"
      args = ["install", "."]
      env  = ["PROJECT_ROOT=${var.namespace}"]
    }

    step { #A
      name = "gcr.io/cloud-builders/go"
      args = ["test"]
      env  = ["PROJECT_ROOT=${var.namespace}"]
    }

    step { #A
      name = "gcr.io/cloud-builders/docker"
      args = ["build", "-t", local.image, "."]
    }

    step { #A
      name = "gcr.io/cloud-builders/docker"
      args = ["push", local.image]
    }

    step { #A
      name = "gcr.io/cloud-builders/gcloud"
      args = ["run", "deploy", google_cloud_run_service.service.name, "--image", local.image, "--region", var.region, "--platform", "managed", "-q"]
    }
  }
}
```

#A there are lots of repeating configuration blocks for the steps in the build process

As you can see, this works, but it's not a flexible solution. First, the steps in the build process must be statically declared in Terraform, which doesn't help if you want to dynamically generate build steps or pass the build steps as an input variable (such as for a customer facing module). Moreover, if you don't know the steps in the build process ahead of time, then declaring your build process in Terraform is too rigid and cumbersome. To solve this annoying problem, HashiCorp introduced a new expression called *dynamic blocks*.

### 7.4.5 Dynamic Blocks: Rare Boys

Dynamic blocks are the least used of all Terraform expressions, and for good reason. They were designed to solve the niche problem of how to dynamically create nested configuration blocks in Terraform. Dynamic blocks can *only* be used within other blocks, and *only* when the use of repeatable nested configuration blocks is supported (surprisingly uncommon). Nevertheless, dynamic blocks are invaluable if you find yourself in a situation where you need one, such as creating rules for a security group, or steps in a Cloud Build workflow.

Dynamic nested blocks act much like *for expressions* but produce nested configuration blocks instead of complex types. They iterate over complex types (such as maps and lists) and generate configuration blocks for each element. Syntax of a dynamic nested block is presented in Figure 7.13

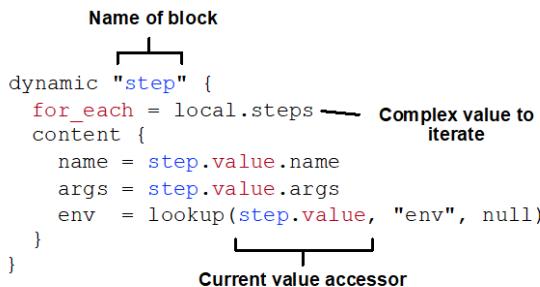


Figure 7.13 Syntax for a dynamic nested block

**WARNING** Overuse of dynamic nested blocks makes your code significantly harder to understand, so use them sparingly.

Typically, dynamic nested blocks are combined with local values or input variables (because otherwise your code would be statically defined, and you wouldn't need a dynamic block). In our case it doesn't matter (since we aren't parameterizing the build steps anyways) but it is good practice. For such local blocks that serve only as helpers, I like to declare them right above where they are being used. This makes them easier to find, as they do not benefit from being at the top of the file. Please append the contents of Listing 7.6 to your `main.tf` for provisioning the Cloud Build trigger and the steps it will use to run.

#### Listing 7.6 main.tf

```

locals { #A
  image = "gcr.io/${var.project_id}/${var.namespace}"
  steps = [
  {
    name = "gcr.io/cloud-builders/go"
    args = ["test"]
  }
]
  
```

```

env = ["PROJECT_ROOT=${var.namespace}"]
},
{
  name = "gcr.io/cloud-builders/docker"
  args = ["build", "-t", local.image, "."]
},
{
  name = "gcr.io/cloud-builders/docker"
  args = ["push", local.image]
},
{
  name = "gcr.io/cloud-builders/gcloud"
  args = ["run", "deploy", google_cloud_run_service.service.name, "--image", local.image, "--region", var.region, "--platform", "managed", "-q"]
}
]

}
}

resource "google_cloudbuild_trigger" "trigger" {
depends_on = [
  google_project_service.enabled_service["cloudbuild.googleapis.com"]
]

trigger_template {
  branch_name = "master"
  repo_name  = google_sourcerepo_repository.repo.name
}

build {
  dynamic "step" {
    for_each = local.steps
    content {
      name = step.value.name
      args = step.value.args
      env = lookup(step.value, "env", null) #B
    }
  }
}
}
}

```

#A declaring local values right before using them helps with readability  
#B not all steps have “env” set. This sets content.env to null if step.value[“env”] is not set

Before we move on to the next section, let’s add some IAM related configuration to `main.tf`. Listing 7.7 shows the code for adding two roles to the Cloud Build IAM service account, which enables Cloud Build to deploy services to Cloud Run during a `terraform apply`.

#### **Listing 7.7 main.tf**

```

data "google_project" "project" {}

resource "google_project_iam_member" "cloudbuild_roles" {
depends_on = [google_cloudbuild_trigger.trigger]
for_each  = toset(["roles/run.admin", "roles/iam.serviceAccountUser"])#A
}

```

```

project  = var.project_id
role     = each.key
member   = "serviceAccount:${data.google_project.project.number}@cloudbuild.gserviceaccount.com"
}

#A grant the Cloud Build service account these two roles

```

## 7.5 Configuring a Serverless Container

Now we need to configure the Cloud Run service for running our serverless container after it has been deployed with Cloud Build. There are two steps to this process. First, we need to declare and configure the Cloud Run service itself; second, we need to explicitly enable unauthenticated user access (because the default is Deny All).

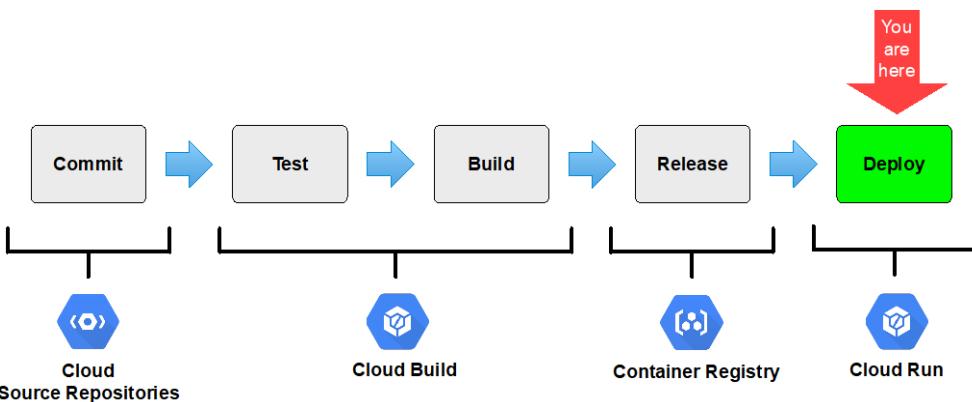


Figure 7.14 CI/CD pipeline - Stage 3 of 3

The code for configuring the Cloud Run service is shown in Listing 7.8. It's simple enough, we declare that we want to use the latest version of the image from the Container Registry to start a single container. Please add this to your `main.tf`.

**NOTE** due to an implementation detail of Cloud Run, new services throw an error during initial creation if there is not at least one image available in the Container Registry. This means we need to publish an image before `terraform apply` will succeed

### Listing 7.8 main.tf

```

resource "google_cloud_run_service" "service" {
  depends_on = [
    google_project_service.enabled_service["run.googleapis.com"]
  ]
  name      = var.namespace
  location  = var.region
}

```

```
template {
  spec {
    containers {
      image = "${local.image}:latest" #A
    }
  }
}
```

#A the Cloud Run service will use the latest version of the image from the Container Registry

To expose the web application to the internet, we need to enable unauthenticated access. We can do that with an IAM policy that grants all users the `run.invoker` role to the provisioned Cloud Run service. Again, add the code from the following listing to the bottom of `main.tf`.

#### **Listing 7.9 main.tf**

```
data "google_iam_policy" "admin" {
  binding {
    role = "roles/run.invoker"
    members = [
      "allUsers",
    ]
  }
}

resource "google_cloud_run_service_iam_policy" "policy" {
  location  = var.region
  project   = var.project_id
  service   = google_cloud_run_service.service.name
  policy_data = data.google_iam_policy.admin.policy_data
}
```

There's just a couple of minor things we need to do before finishing up this scenario: create an `outputs.tf` and a `versions.tf`. The `outputs.tf` file will reference the URLs from the source repository and Cloud Run service. We will need both of these later. Create an `outputs.tf` with the following content:

#### **Listing 7.10 outputs.tf**

```
output "urls" {
  value = {
    repo = google_sourcerepo_repository.repo.url
    app = google_cloud_run_service.service.status[0].url
  }
}
```

Finally, create a `versions.tf` to lock in the GCP provider version.

#### **Listing 7.11 versions.tf**

```
terraform {
```

```
required_version = "~> 0.12"
required_providers {
  google  = "~> 3.10"
}
}
```

## 7.6 Deploying Static Infrastructure

Remember that there are two parts to this project, the static (Terraform) part and the dynamic (non-Terraform) part. What we have been working on so far only amounts to the static part, which is responsible for laying down all the underlying infrastructure that the dynamic component needs to run. We will talk how to deploy the dynamic component in the next section. For now, the complete source code of `main.tf` is shown in Listing 7.12, for your convenience:

**Listing 7.12 complete main.tf**

```
locals {
  services = [
    "sourcerepo.googleapis.com",
    "cloudbuild.googleapis.com",
    "run.googleapis.com",
    "iam.googleapis.com",
  ]
}

resource "google_project_service" "enabled_service" {
  for_each = toset(local.services)
  project = var.project_id
  service = each.key

  provisioner "local-exec" {
    command = "sleep 60"
  }

  provisioner "local-exec" {
    when  = destroy
    command = "sleep 15"
  }
}

resource "google_sourcerepo_repository" "repo" {
  depends_on = [
    google_project_service.enabled_service["sourcerepo.googleapis.com"]
  ]

  name = "${var.namespace}-repo"
}

locals {
  image = "gcr.io/${var.project_id}/${var.namespace}"
  steps = [
    {
      name = "gcr.io/cloud-builders/go"
      args = ["test"]
    }
  ]
}
```

```

env = ["PROJECT_ROOT=${var.namespace}"]
},
{
  name = "gcr.io/cloud-builders/docker"
  args = ["build", "-t", local.image, "."]
},
{
  name = "gcr.io/cloud-builders/docker"
  args = ["push", local.image]
},
{
  name = "gcr.io/cloud-builders/gcloud"
  args = ["run", "deploy", google_cloud_run_service.service.name, "--image", local.image, "--region", var.region, "--platform", "managed", "-q"]
}
]
}

resource "google_cloudbuild_trigger" "trigger" {
  depends_on = [
    google_project_service.enabled_service["cloudbuild.googleapis.com"]
  ]
}

trigger_template {
  branch_name = "master"
  repo_name = google_sourcerepo_repository.repo.name
}

build {
  dynamic "step" {
    for_each = local.steps
    content {
      name = step.value.name
      args = step.value.args
      env = lookup(step.value, "env", null)
    }
  }
}
}

data "google_project" "project" {}

resource "google_project_iam_member" "cloudbuild_roles" {
  depends_on = [google_cloudbuild_trigger.trigger]
  for_each = toset(["roles/run.admin", "roles/iam.serviceAccountUser"])
  project = var.project_id
  role = each.key
  member = "serviceAccount:${data.google_project.project.number}@cloudbuild.gserviceaccount.com"
}

resource "google_cloud_run_service" "service" {
  depends_on = [
    google_project_service.enabled_service["run.googleapis.com"]
  ]
  name = var.namespace
  location = var.region
}

```

```

template {
  spec {
    containers {
      image = "${local.image}:latest"
    }
  }
}

data "google_iam_policy" "admin" {
  binding {
    role = "roles/run.invoker"
    members = [
      "allUsers",
    ]
  }
}

resource "google_cloud_run_service_iam_policy" "policy" {
  location  = var.region
  project   = var.project_id
  service   = google_cloud_run_service.service.name
  policy_data = data.google_iam_policy.admin.policy_data
}

```

**NOTE** This code won't run! `terraform apply` will fail if there isn't already an image in the Container Registry. This is due to some weirdness in the Cloud Run API, which isn't handled properly by the GCP provider. A workaround is presented in the next section.

When you're ready, initialize and deploy the infrastructure to GCP:

```
$ terraform init && terraform apply -auto-approve
...
google_project_iam_member.cloudbuild_roles["roles/iam.serviceAccountUser"]: Creation complete after 10s [id=tic-pipelines/roles/iam.serviceAccountUser/serviceaccount:783629414819@cloudbuild.gserviceaccount.com]
```

Apply complete! Resources: 10 added, 0 changed, 0 destroyed.

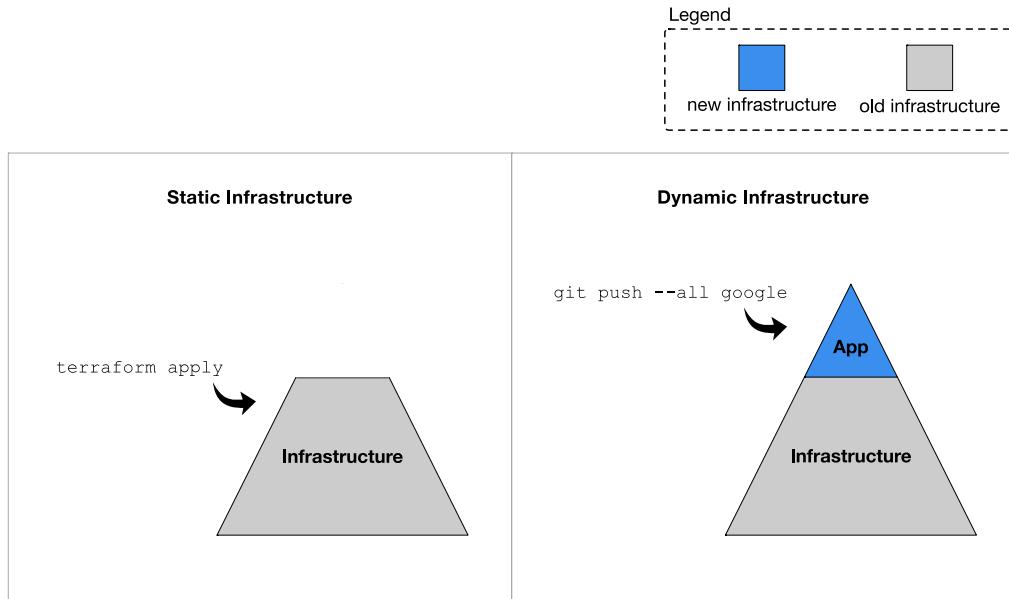
Outputs:

```
urls = {
  "app" = "https://team-pokemon3-udm3wrbjga-uc.a.run.app"
  "repo" = "https://source.developers.google.com/p/terraformaction/r/pokemon-repo"
}
```

## 7.7 CI/CD of a Docker Container

Now we're going to deploy a container through the CI/CD pipeline. I will also present the workaround I mentioned to get the previous infrastructure code deployed successfully. The code in this section is not that important, so don't overthink it. Basically, we're going to Dockerize a simple golang server for listening on port 8080 and serving the string: "Pikachu!". This is analogous to a simple "Hello World!" app. To give you some context, this app could be a stand-

in for the super cool app that the development team, “Team Pokémon” is working on. After all, it’s always a good idea to test the CI/CD pipeline workflow before handing it over to the team that will be using it.



**Figure 7.15 dynamic infrastructure is deployed on top of the static infrastructure**

You should have two folders, “application” and “infrastructure”. All the code up until now should be in the “infrastructure” folder. To get started with the application code, switch over to the application folder:

```
$ cd ./application
```

In this directory, create a `main.go` file which will be the main entry point for our server:

#### **Listing 7.13 main.go**

```
package main

import (
    "fmt"
    "log"
    "net/http"
)

func IndexServer(w http.ResponseWriter, r *http.Request) {
    fmt.Fprint(w, "Pikachu!")
}

func main() {
```

```

    handler := http.HandlerFunc(IndexServer)
    log.Println("Listening on :8080...")
    http.ListenAndServe(":8080", handler) #A
}

#A starts the server on port 8080 and serves the string: "Pikachu!"
```

Next, write a super basic test file `main_test.go` with the following:

#### **Listing 7.14 main\_test.go**

```

package main

import (
    "net/http"
    "net/http/httpptest"
    "testing"
)

func TestGETIndex(t *testing.T) {
    t.Run("returns index", func(t *testing.T) {
        request, _ := http.NewRequest(http.MethodGet, "/", nil)
        response := httpptest.NewRecorder()

        IndexServer(response, request)

        got := response.Body.String()
        want := "Pikachu!"

        if got != want {
            t.Errorf("got '%s', want '%s'", got, want)
        }
    })
}
```

Then, create a `Dockerfile` for packaging the application. Listing 7.15 shows the code for a basic multi-stage `Dockerfile` that will work for our purposes.

#### **Listing 7.15 Dockerfile**

```

FROM golang:1.12 as builder
WORKDIR /go/src/github.com/team-pokemon/pikachu
COPY . .
RUN CGO_ENABLED=0 GOOS=linux go build -v -o app

FROM alpine
RUN apk update && apk add --no-cache ca-certificates
COPY --from=builder /go/src/github.com/team-pokemon/pikachu/app /app
CMD ["/app"]
```

### 7.7.1 Publishing to Container Registry

Let's take a short intermission from the main scenario to fix the problem we had earlier with the infrastructure code being unable to deploy. As a reminder, the infrastructure was unable to deploy because of an error with Cloud Run service that prevents it from running if there is not at least one container in the Container Registry. We can solve that by building and publishing an initial image now.

**NOTE** this is a one-time setup operation that needs to take place before the CI/CD infrastructure can be deployed. This would not be part of a normal developer's workflow

First build the docker container with the following command. You will need to replace `project_id` with your GCP project id, and `namespace` with whatever the value of your namespace is (probably `team-pokemon`):

```
$ docker build -t gcr.io/<project_id>/<namespace>.
Sending build context to Docker daemon 60.93kB
Step 1/8 : FROM golang:1.12 as builder
--> ffcabee6f7d8b
Step 2/8 : WORKDIR /go/src/github.com/team-pokemon/pikachu
--> Using cache
--> b8d6fa3d4316
Step 3/8 : COPY ..
--> Using cache
--> 4b31d4bd8964
Step 4/8 : RUN CGO_ENABLED=0 GOOS=linux go build -v -o app
--> Using cache
--> afd8bc5c568a
Step 5/8 : FROM alpine
--> e7d92cdc71fe
Step 6/8 : RUN apk update && apk add --no-cache ca-certificates
--> Using cache
--> 44bb5c35b734
Step 7/8 : COPY --from=builder /go/src/github.com/team-pokemon/pikachu/app /app
--> Using cache
--> 4d077d10232c
Step 8/8 : CMD ["/app"]
--> Using cache
--> 52645d2675e3
Successfully built 52645d2675e3
Successfully tagged gcr.io/terraformaction/team-pokemon:latest
```

Next, obtain authentication credentials for publishing to the Container Registry (output not shown):

```
$ gcloud auth login && gcloud auth configure-docker -q
```

Then, push the previously build image to the Container Registry (you may need to enable the Google Container Registry API as well – there will be a link to do so if the command fails):

```
$ docker push gcr.io/<project_id>/<namespace>
The push refers to repository [gcr.io/terraformaction/team-pokemon]
```

```
f7db561234da: Pushed
9e08da152dd7: Pushed
5216338b40a7: Pushed
latest: digest: sha256:25c50ac1e7a1b8f80ac9b00f05d40893ea0a6d6af733bc5759ae1c73836d8500 size: 950
```

Finally, switch back to the infrastructure folder and deploy the code with a terraform apply. It will succeed this time.

```
$ terraform apply -auto-approve
...
google_project_iam_member.cloudbuild_roles["roles/iam.serviceAccountUser"]: Creation complete after 10s [id=tic-pipelines/roles/iam.serviceAccountUser/serviceaccount:783629414819@cloudbuild.gserviceaccount.com]

Apply complete! Resources: 10 added, 0 changed, 0 destroyed.

Outputs:

urls = {
  "app" = "https://team-pokemon3-udm3wrbjga-uc.a.run.app"
  "repo" = https://source.developers.google.com/p/terraformaction/r/pokemon-repo
}
```

## 7.7.2 Test Run the CI/CD Pipeline

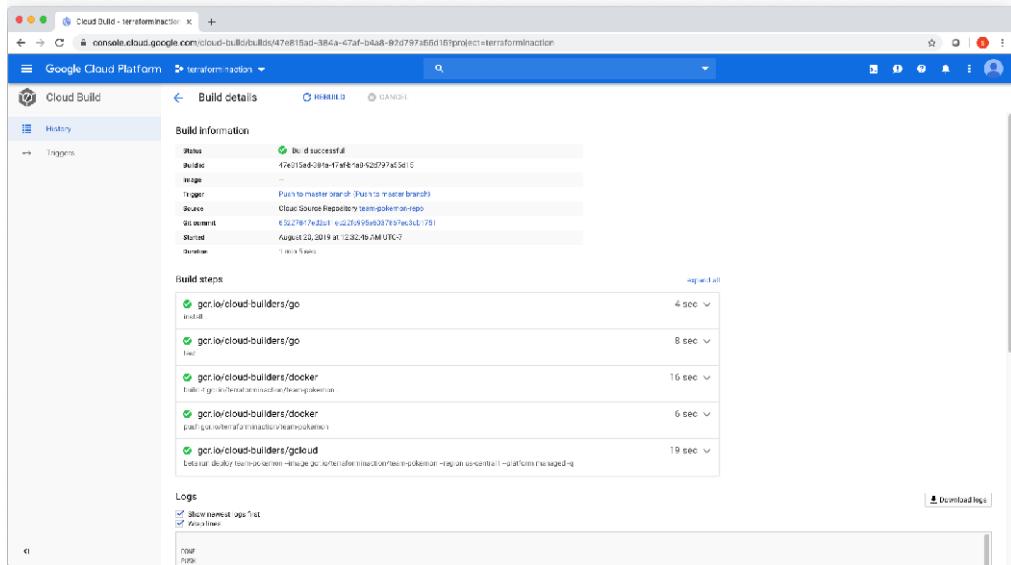
Now that the CI/CD pipeline is up and running, go back to the application directory, initialize Git, stage all changes, and push everything to the source repository. This will trigger Code Build to check out the source code, build the image, publish to the Container Registry, and deploy a new revision to Cloud Run. Listing 7.16 shows the commands necessary to make this happen. You'll need to substitute in the repo URL from the Terraform outputs.

### **Listing 7.16 Git commands**

```
git init && git add -A && git commit -m "initial push"
git config --global credential.helper https://source.developers.google.com.helper gcloud.sh
git remote add google <urls.repo> #A
gcloud auth login && git push --all google
```

#A insert your source repo url here

After you've pushed your code, you can view the status of the build in the Cloud Build console. Figure 7.16 shows an example of what an in-progress deployment might look like.



**Figure 7.16** Cloud Build triggers a build when you commit to the master branch. This will build, test, publish and finally deploy the code to Cloud Run.

When the build completes, you can navigate to the service URL in the browser (from the “app” output attribute). You should see a spartan website with the word “Pikachu!” emblazoned in plain text (see figure 7.17). This means you successfully completed the scenario.



**Figure 7.17** Example deployed website

**WARNING!** Don’t forget to clean up your static infrastructure with a `terraform destroy` to avoid paying ongoing costs!

## 7.8 Fireside Chat

Give yourself a pat on the back, this chapter has been incredibly technically challenging. Hopefully, you’ve learned some valuable tips and tricks for those times when Terraform just doesn’t do what you need it to do.

**WARNING!** Backdoors to Terraform (i.e. local-exec provisioners) are inherently dangerous and should be avoided whenever possible. Use them only as a means of last resort.

We started by talking about two-stage deployments, whereby Terraform is used to deploy static infrastructure, and a CI/CD pipeline is used to deploy dynamic infrastructure on top of that. By structuring your project in such a way, you're able to maintain a separation of duties between "what changes a lot" and "what changes a little".

Our static infrastructure was the CI/CD pipeline. Despite being "static", the Terraform code we saw was actually extremely dynamic and expressive, allowing us to do all sorts of cool stuff with For-Each, dynamic blocks and resource provisioners.

Although we only talked about local-exec provisioners, there are actually a few different kinds of resource provisioners. Table 7.1 outlines a reference of all the provisioners that exist in Terraform.

**WARNING!** HashiCorp thinks resource provisioners are bad practice and have hinted they may deprecate their use in a later version of Terraform!

**Table 7.1** reference of all resource provisioners in Terraform

Name	Description	Example
chef	Installs, configures and runs the Chef Client on a remote resource	<pre>provisioner "chef" {   attributes_json = file("chef.json") }</pre>
file	Copies files or directories from the machine executing Terraform to the newly created resource	<pre>provisioner "file" {   source    = "conf/myapp.conf"   destination = "/etc/myapp.conf" }</pre>
habitat	Installs the Habitat supervisor and loads configured services	<pre>provisioner "habitat" {   peer = aws_instance.ex.private_ip   use_sudo = true   service_type = "systemd"    service {</pre>

		<pre> name = "core/redis" topology = "leader" user_toml = file("redis.toml") } } </pre>
local-exec	Invokes an arbitrary process on the machine running Terraform (not on the resource)	<pre> provisioner "local-exec" {   command = "echo hello" } </pre>
remote-exec	Invokes a script on a remote resource after it is created. This can be used to run configuration management tools, bootstrap scripts, etc.	<pre> provisioner "remote-exec" {   inline = [     "puppet apply",   ] } </pre>
salt-masterless	Provisions machines using Salt states, without connecting to a Salt master	<pre> provisioner "salt-masterless" {   "local_state_tree" = "/srv/salt" } </pre>

## 7.9 Summary

- Design and deploy an end-to-end CI/CD pipeline for Docker containers on GCP
- Compare and contrast two methods for continuous deployment with Terraform:
  - By packaging and shipping the application with configuration code
  - By isolating the application from the underlying infrastructure and performing continuous deployment through a separate mechanism
- For-Each can be used to provision resources dynamically as a substitute for count. Likewise, dynamic blocks allow you to generate repeating configuration blocks. Only use fancy dynamic expressions if you really need them, they make your code harder to read.
- Providers can be explicitly passed to resources and modules. Explicit providers always

- override implicit providers.
- Null resources can be combined with local-exec provisioners to create custom resources.  
Another way to create custom resources is with the Shell provider.

# 8

## A Multi-Cloud MMORPG

### This chapter covers:

- Deploying a multi-cloud load balancer for distributing application traffic across virtual machines running in AWS, Azure and GCP
- Federating two Nomad clusters, one in AWS and one in Azure, for managing multi-cloud workloads with a single control plane
- Running a browser-based MMORPG with Nomad
- Rearchitecting multi-cloud projects using a mixture of managed services

Terraform is your all-in-one multi-cloud solution! Terraform allows you to dynamically provision infrastructure across multiple cloud providers, so your workflows and processes can remain the same even as the world around you changes. This chapter puts into practice all the learnings from previous chapters to give you a holistic perspective of what is possible with Terraform.

First, however, I must define what is meant by “multi-cloud”. Although often used interchangeably, the terms “hybrid cloud” and “multi-cloud” are not equivalent and should not be conveyed as such. *Multi-cloud* specifically means leveraging multiple cloud providers in a single context. *Hybrid cloud*, on the other hand, is distinct in that it refers to the union of a private cloud with one or more public clouds. While hybrid cloud could be considered a kind of multi-cloud, rarely is this done. This is because the implied level of maturity in the cloud and the sorts of problems one could normally be expected to face are different when comparing the hybrid versus the multi-cloud.

Multi-cloud is widely considered the more cutting-edge and sophisticated of the two, largely because most organizations pursue it only after already having experience with the hybrid cloud. It’s an organic process. Much like the growth of a tree, an organization’s journey to the cloud first begins with nothing but the seed of a dream. After an organization becomes more fluent

with the cloud, they may begin migrating production workloads onto the cloud. Just as a sapling represents a snapshot in the life cycle of a tree, so too does the hybrid cloud represent a state of transition from private cloud to multi-cloud. The goal is not to be in this state of transition indefinitely. Multi-cloud is the natural terminus – but requires a high level of maturity to achieve.

While organizations may choose multi-cloud for a multitude of reasons, the most commonly stated reasons are:

1. **Flexibility** – No one provider can be everything to everyone. By adopting a multi-cloud strategy, you have the ability to pick and choose best-in-class services
2. **Cost Savings** – Even for the same kinds of services, pricing models can vary drastically between cloud providers. You can save a lot of money by optimizing for cost in the multi-cloud. Some of these cost savings may be counterbalanced by increased operational complexity, however.
3. **Avoiding Vendor Lock-in** – By architecting for the multi-cloud, your business stays nimble and can avoid being locked into proprietary systems and protocols. Note that avoiding vendor lock-in isn't a guaranteed thing if a poor architecture is chosen. Particularly, if cloud specific services are chosen rather than cloud agnostic technologies, such as SQL, K8S, object stores and so on, you could find yourself locked into two clouds rather than just one.
4. **Resilience** – Multiple redundant systems can make your business less susceptible to DDoS attacks and SPOF (single point of failure) incidents. It can also reduce downtime and protect you from service outages.
5. **Compliance** – To meet governance, risk management, and SLA policies, you may be required to adopt a multi-cloud strategy whether you like it or not. For example, you may be required to use AliCloud to comply with government regulations for operating in China.

Despite the many clear benefits of multi-cloud, nothing comes for free. The major downsides of adopting multi-cloud include:

1. **Operational Complexity** – Instead of supporting one public cloud, now you're supporting two or more. This means you need domain expertise in these other clouds, which often means hiring more engineers.
2. **Security Concerns** – It is hard to manage and audit data when it's moving between multiple clouds. There is greater potential for compromises of security that could lead to data breaches.
3. **Architectural Complexity** – Few people have the depth of knowledge necessary to design a seamless multi-cloud experience. Even if you can design such a system, can it be maintained throughout the entire Software Development Life Cycle (SDLC)?

Ultimately, you should decide for yourself whether multi-cloud is right for you. It is unlikely that many small to medium sized businesses will find it worth it, if only because their limited resource pool would be better spent elsewhere. Nevertheless, many industry experts agree that the future

is moving toward multi-cloud, so preparing a multi-cloud strategy early may be the prudent course of action.

If you are going to brainstorm a multi-cloud narrative, the value of orchestration and automation absolutely cannot be overstated. There are so many moving parts in multi-cloud that any manual process would quickly fall flat on its face. This is where Terraform fits in. Terraform is the perfect tool for multi-cloud orchestration and automation because it allows you to seamlessly deploy to multiple clouds with a single `terraform apply`. You may wish to combine Terraform with other technologies or processes, such as configuration management tools, or CI/CD pipelines, but Terraform is more than capable by itself – as we shall soon see.

In this chapter we will investigate several approaches for architecting multi-cloud projects. First, we will deploy a hybrid cloud load balancer that distributes traffic evenly to virtual machines located in AWS, Azure and GCP. This example exemplifies how easy it is to get started with hybrid/multi-cloud deployments when you already know Terraform.

Next is my favorite part. We'll deploy the infrastructure and containerized services necessary to run *BrowserQuest*, a Massively Multiplayer Online Role-Playing Game (MMORPG) created by Mozilla that's playable in the browser. This game is surprisingly entertaining, and I wasted a whole afternoon with a coworker when I first deployed it. A preview of BrowserQuest is shown in figure 8.1.



Figure 8.1 *BrowserQuest* is a massively multiplayer HTML5 game that you can play through the browser

We will deploy the MMORPG application in two ways, first by using containers on a *federated* container orchestration platform (i.e. Nomad), and again by using managed cloud services.

**DEFINITION** A *federation* is a group of computing or network providers agreeing upon standards of operation in a collective fashion. The term may be used when describing the inter-operation of two distinct, formally disconnected, telecommunications networks that may have different internal structures.

The reason for solving the same problem twice is to emphasize the fact that there isn't one "right" way to multi-cloud. Because of the increased flexibility you get, you can choose whatever makes the most sense for you and your business needs. This may mean creating a single API that abstracts away the differences between the clouds (as is the case with a federated container orchestration platform), or it may mean embracing what makes each cloud unique. Large organizations may benefit from the consistency that containers and container orchestration platforms grant you, while smaller organizations may benefit from the speed of delivery and cost savings that managed services provide. Either way, the choice is yours to make.

## 8.1 Hybrid Cloud Load Balancing

We are going to start by adding a twist to the classic load balancer problem. Specifically, we're going to deploy a hybrid cloud load balancer that combines resources on your local machine (a.k.a the "private cloud" for the purposes of this exercise) with virtual machines hosted in AWS, Azure and GCP. The end result will be a simple HTML/CSS website that lets you know what cloud the connected server is running on. When you refresh the page, the load balancer will send you to the next server, and the next server, and so on, using round-robin DNS. This can be visualized by figure 8.2

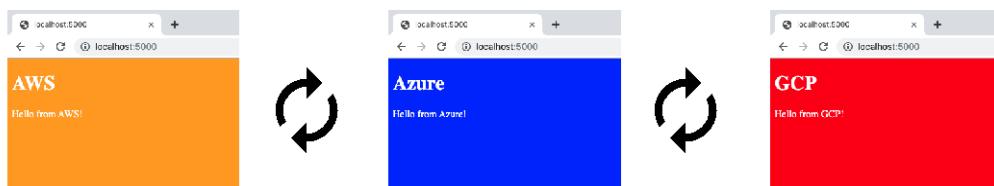
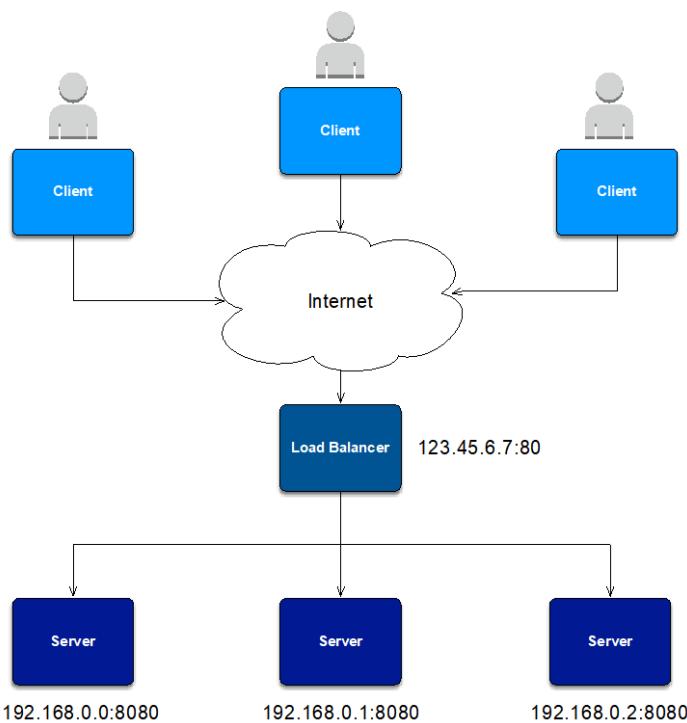


Figure 8.2 Each time the page is refreshed, it displays text and color to let you know what cloud the server is running on

The purpose of this exercise is not to suggest that you should replicate this in a production environment, but to give you some ideas of what is possible and show you just how easy it is to get started with multi-cloud projects in Terraform. It's not as scary as it might seem, and hopefully you will find that it is not much different than what we've doing all along.

### 8.1.1 Architectural Overview

As a reminder, the role of a load balancer is to distribute internet traffic across multiple servers, to improve both the reliability and scalability of applications. As servers come and go, the load balancer automatically routes traffic to healthy instances based on various routing rules, while still providing a single static point of entry to users. Typically, all instances that make up the server farm will be collocated and networked together on the same private network (see figure 8.3).

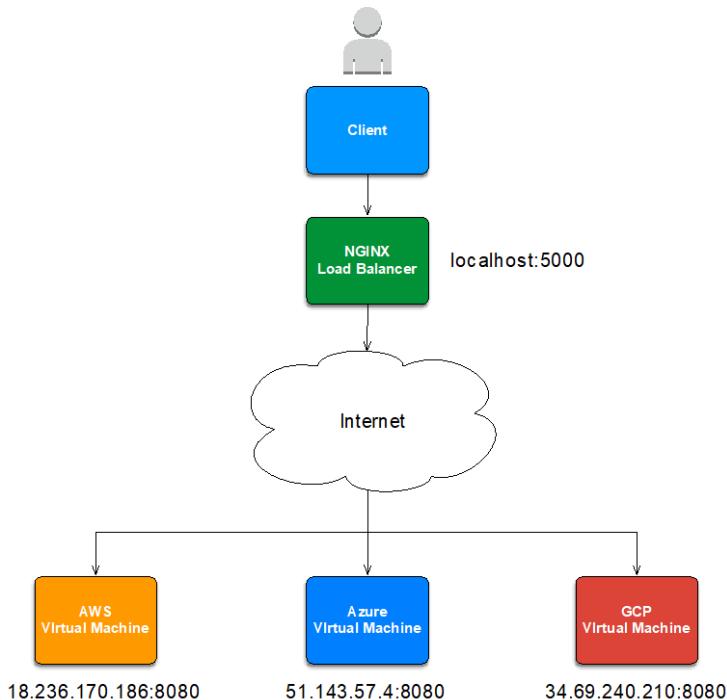


**Figure 8.3 A classic load balancer setup. Clients talk to the load balancer over the internet, and all the servers behind the load balancer are on the same private network**

By contrast, the hybrid cloud load balancer architecture (see figure 8.4) will seem unconventional, but I shall explain. First, each server lives in a separate cloud and is assigned a public IP that is used to register itself with the load balancer.

**NOTE** Normally you would not assign a public IP address to instances behind a load balancer, but I did it this way to simplify the problem for the purposes of this exercise.

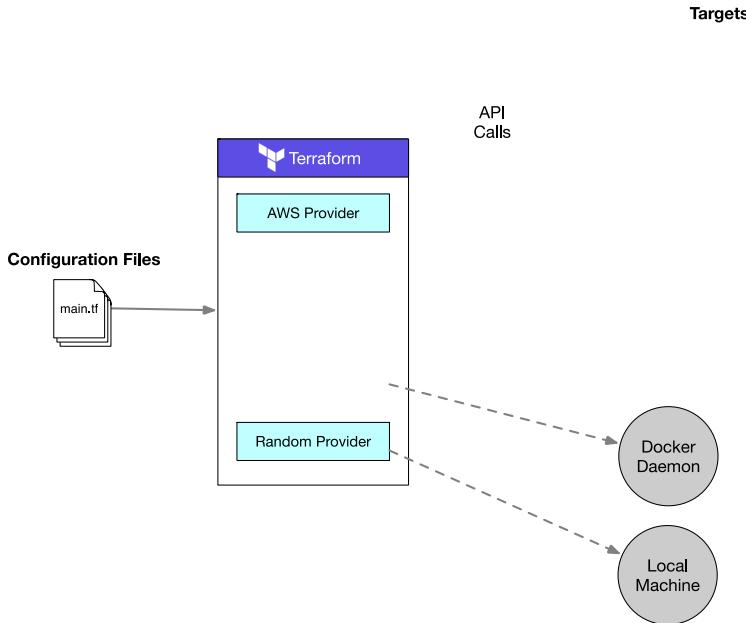
Another oddity is that the load balancer itself runs as a docker container on the local machine, mapped to loopback port 5000. This one is not as weird as it seems, because you could configure DNS/firewall rules to allow traffic to be served on the internet (remember your local machine counts as a kind of “private cloud”!). It also permits an excuse to introduce the Docker provider for Terraform.



**Figure 8.4 Hybrid cloud load balancing with a private cloud load balancer and public cloud virtual machines**

**NOTE** in production, I recommend using private network tunnels between the clouds so that the servers are not exposed publicly over the internet

We will be using five distinct providers in this scenario. Three of them (AWS, Azure and GCP) are cloud providers, and the other two (Random and Docker) deploy resources only to the local machine. This can be visualized by figure 8.5.



**Figure 8.5** the scenario uses five distinct Terraform providers to deploy various resources to public and private cloud targets

### 8.1.2 Code

The configuration code for this scenario is short and easily understandable, because a majority of the business logic is encapsulated in modules. I did this because the code is simply too long to fit alongside all the other material I want to cover. Don't worry though, you aren't missing anything with this abridged edition; there is nothing in these modules that you shouldn't already know how to do by now. Of course, if you are interested, you can always poke around the modules to see makes them tick.

Let's begin by creating a `variables.tf` containing three variables, one to configure each of the cloud providers.

**TIP** this scenario also works well with fewer than three clouds. If you choose not to deploy to all three clouds, simply comment out all the configuration code and references to the undesired provider(s) in this and subsequent code listings.

#### **Listing 8.1** `variables.tf`

```
variable "aws" {
  type = object({
```

```

profile = string
region = string
})
}

variable "azure" {
type = object({
subscription_id = string
client_id     = string
client_secret  = string
tenant_id     = string
location      = string
})
}

variable "gcp" {
type = object({
project_id = string
region    = string
})
}

```

We will use the declared variables to configure each of the three major cloud providers (see Listing 8.2). Note that the Docker provider does not need to be configured with authentication information, because it will be directly accessing the Docker daemon over an unauthenticated port running on localhost.

### The Curious Docker Provider for Terraform

Once you've been indoctrinated into the wonderful world of Terraform, it's only natural to want Terraform to do everything. After all, why not have more of a good thing? Unfortunately, Terraform is not the right solution for all problems. Being a Terraform guru means having the wisdom to know that just because you can do something doesn't necessarily mean you should. The Terraform provider for Docker is one such example of something that sounds better on paper than it is in practice.

I've always found the Docker CLI commands or Docker Compose to be easier to use and less buggy than the Docker provider for Terraform. It may even be better to wrap these commands with a Shell provider or local-exec provisioner than rely on the official providers. In my experience, the Docker provider – alongside other similar providers – are interesting curiosities, but not production ready. That being said, it is still good to know how to use these kinds of providers.

### **Listing 8.2 providers.tf**

```

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

provider "azurerm" {
subscription_id = var.azure.subscription_id
client_id     = var.azure.client_id

```

```

client_secret = var.azure.client_secret
tenant_id    = var.azure.tenant_id
}

provider "google" {
  credentials = file("account.json")
  project    = var.gcp.project_id
  region     = var.gcp.region
}

provider "docker" {
}

```

**TIP** feel free to configure the providers however you like. Just because I do it one way doesn't mean you have to do it exactly the same. Just remember to update the variables and provider declarations as necessary.

Next we need to set the variable values in `terraform.tfvars`. You will also need to fill in the blanks with your specific secrets, such as which AWS profile and region to use, the static Azure credentials, and the GCP project\_id. As discussed in the setup for chapter 7, the GCP project needs to be created ahead of time, and you need to have an `account.json` file or authenticate using the console<sup>1</sup>.

### Listing 8.3 `terraform.tfvars`

```

aws = {
  profile = "<profile>"
  region  = "us-west-2"
}

azure = {
  subscription_id = "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxx"
  client_id      = "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxx"
  client_secret   = "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxx"
  tenant_id      = "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxx"
  location       = "centralus"
}

gcp = {
  project_id = "<project_id>"
  region    = "us-east1"
}

```

Now here's the juicy bit, the part that deploys virtual machines to AWS, Azure and GCP, along with the NGINX load balancer. I will admit it may come across as "hand wavy" because of the heavy reliance on modules, but again, I did this to save space and because the details aren't the main teaching point of this exercise.

<sup>1</sup> [https://www.terraform.io/docs/providers/google/provider\\_reference.html#full-reference](https://www.terraform.io/docs/providers/google/provider_reference.html#full-reference)

**NOTE** the module source path in Listing 8.4 is atypical and has an extra slash in the middle. This is a trick to refer to individual submodules within a monolithic GitHub repo, which is convenient for small and/or related modules.

#### Listing 8.4 main.tf

```
module "aws" {
  source = "scottwinkler/vm/cloud//modules/aws" #A
  environment = {
    name = "AWS"
    background_color = "orange"
  }
}

module "azure" {
  source = "scottwinkler/vm/cloud//modules/azure" #A
  environment = {
    name = "Azure"
    background_color = "blue"
  }
}

module "gcp" {
  source = "scottwinkler/vm/cloud//modules/gcp" #A
  project_id = var.gcp.project_id
  environment = {
    name = "GCP"
    background_color = "red"
  }
}

module "loadbalancer" {
  source = "scottwinkler/vm/cloud//modules/loadbalancer" #A
  addresses = [
    module.aws.network_address, #C
    module.azure.network_address,#C
    module.gcp.network_address, #C
  ]
}
```

#A All of these modules live in the same mono-repo

#B the environment attribute customizes the name and background color of the webpage

#C network\_address is an output of each on the VM modules and represents the publicly facing URL. It is required by the load balancer to configure routing rules .

Outputs are shown in listing 8.5. These outputs are here for convenience, so you don't have dig through contents of the state file to find the values.

#### Listing 8.5 outputs.tf

```
output "addresses" {
  value = {
    aws      = module.aws.network_address
```

```

azurerm = module.azure.network_address
gcp     = module.gcp.network_address
loadbalancer = module.loadbalancer.network_address
}
}

```

Lastly, write the Terraform settings to `versions.tf` as presented in listing 8.6.

**TIP** Under no circumstances should you skip creating a `versions.tf`, no matter how much you are tempted! It has saved me more than once from breaking changes in new provider versions.

### Listing 8.6 `versions.tf`

```

terraform {
  required_version = "~> 0.12"
  required_providers {
    aws   = "~> 2.48"
    azurerm = "~> 1.43"
    google = "~> 3.8"
    random = "~> 2.2"
    docker = "~> 2.7"
  }
}

```

### 8.1.3 Deploy

The Docker daemon needs to be started before Terraform can deploy containers with the Docker provider. To understand why, I first need to explain a bit more about the Docker provider and how it works. During initialization, the Docker provider authenticates against the local Docker daemon using an IP address and port over TCP (default is 2375, although this can be configured). If you are using a Unix or Mac system, you can run a Docker container to listen and proxy requests to the Docker daemon with the following command:

```
docker run -d -v /var/run/docker.sock:/var/run/docker.sock -p 127.0.0.1:2375:2375 bobrik/socat TCP-LISTEN:2375,fork UNIX-CONNECT:/var/run/docker.sock
```

If you are using Windows or otherwise having technical difficulties, simply comment out the Docker provider and module entirely from the preceding code and I will show an alternate method in a bit.

**NOTE** although Terraform tries to be platform neutral, configuration that works on one machine isn't guaranteed to work on another, because the local development environment is still relevant.

Once you have this small inconvenience out of the way, you are ready to deploy. First initialize the workspace with a `terraform init` and then run a `terraform apply`.

```
$ terraform apply
...
+ name      = (known after apply)
```

```
+ owner_id      = (known after apply)
+ revoke_rules_on_delete = false
+ vpc_id        = "vpc-e712db81"
}
```

Plan: 19 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?  
Terraform will perform the actions described above.  
Only 'yes' will be accepted to approve.

Enter a value:

After approving and waiting a few minutes, you will get the output addresses for each of the three virtual machines, along with that of the load balancer.

```
module.aws.aws_instance.instance: Creation complete after 16s [id=i-08fcb1592523ebd73]
module.loadbalancer.docker_container.loadbalancer: Creating...
module.loadbalancer.docker_container.loadbalancer: Creation complete after 1s
[id=2e3b541eeb34c95011b9396db9560eb5d42a4b5d2ea1868b19556ec19387f4c2]
```

Apply complete! Resources: 2 added, 0 changed, 0 destroyed.

Outputs:

```
addresses = {
  "aws" = "34.220.128.94:8080"
  "azure" = "52.143.74.93:8080"
  "gcp" = "34.70.1.239:8080"
  "loadbalancer" = "localhost:5000"
}
```

If you don't have the load balancer yet running, you can do so by concatenating the three network addresses with a comma delimiter and directly running the Docker container on your local machine:

```
$export addresses="34.220.128.94:8080,52.143.74.93:8080,34.70.1.239:8080"
$docker run -p 5000:80 -e ADDRESSES=$addresses -dit swinkler/tia-loadbalancer
```

When you navigate to the load balancer address in your browser of choice, you should see the AWS server first. Each time you refresh the page, you will hit a different server. This proves that we have deployed a resilient multi-cloud web application. Even if AWS were to experience a major service outage, the load balancer would automatically route traffic to Azure and GCP, and the site would still be available.



Figure 8.6 An example of the AWS landing page. When you refresh you will get the Azure page (blue), then GCP (red).

**NOTE** it may take a few minutes for all the virtual machines to finish bootstrapping. Keep refreshing until all three appear.

When you are done, remember to clean up with a `terraform destroy`!

```
$ terraform destroy -auto-approve
...
module.gcp.google_compute_instance.compute_instance: Still destroying... [id=gcp-vm, 4m40s elapsed]
module.gcp.google_compute_instance.compute_instance: Still destroying... [id=gcp-vm, 4m50s elapsed]
module.gcp.google_compute_instance.compute_instance: Destruction complete after 4m53s
module.gcp.google_project_service.enabled_service["compute.googleapis.com"]: Destroying... [id=terraform-in-action-lb/compute.googleapis.com]
module.gcp.google_project_service.enabled_service["compute.googleapis.com"]: Destruction complete after 0s

Destroy complete! Resources: 19 destroyed.
```

**WARNING!** if you deployed the Docker container by hand then you need to remember to manually kill it because Terraform isn't doing that for you

## 8.2 Deploying an MMORPG on a Federated Nomad Cluster

Clusters are sets of networked computers that operate as a collective unit much like *The Borg* from the popular TV series, "Star Trek". Clusters are important because they form the backbone behind all container orchestration platforms and make it possible to run highly parallel and distributed workloads at scale. Many companies rely on container orchestration platforms to manage most, if not all, of their production services.

In this section we will deploy a Nomad and Consul cluster in both AWS and Azure; I chose not to include GCP here for the sake of simplicity. Nomad is a general-purpose application scheduler made by HashiCorp that functions as a type of container orchestration platform. Consul, on the other hand, is a service mesh solution (also produced by HashiCorp) and is most similar to Istio<sup>2</sup>.

<sup>2</sup> Istio is a platform-independent service mesh <https://www.istio.io>

**DEFINITION** A *service mesh* is a configurable, low-latency infrastructure layer designed to handle a high volume of network-based interprocess communication among application infrastructure services using APIs

We will only leverage Consul for its ability to enable service-to-service discovery, although it has many other useful capabilities<sup>3</sup>.

The Consul clusters will be federated first. Thereafter, the two Nomad clusters will federate themselves automatically via the SWIM “Gossip” protocol (implemented with Serf<sup>4</sup>). A simplified infrastructure diagram of what we are going to deploy is presented in figure 8.7.

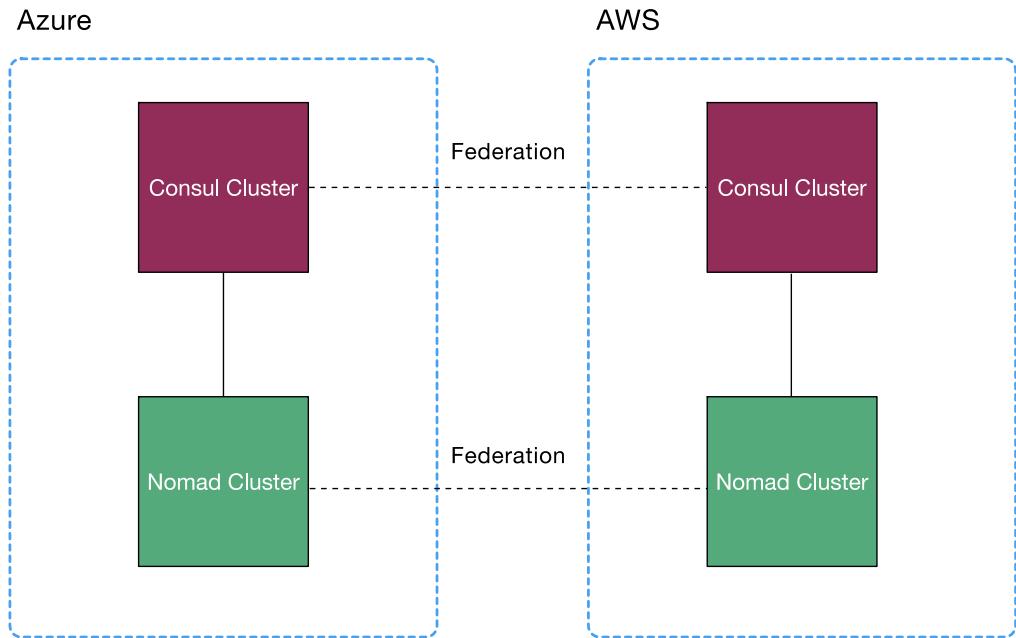


Figure 8.7 The Consul clusters federate themselves together first, and the Nomad clusters federate themselves second

Once the infrastructure is up, we will use the Nomad provider for Terraform to deploy the MMORPG. To illustrate the multi-cloud capabilities of our federated clusters, one container will be running on AWS and another in Azure. At the end of this section we will have a complete and playable multi-cloud video game!

<sup>3</sup> Consul can be configured as a Layer 7 load balancer for automatic service failovers, as an access control enforcer for service policies, and even as a mesh gateway between private networks.

<sup>4</sup> <https://www.serf.io>

### 8.2.1 Cluster Federation 101

Google's Borg paper<sup>5</sup> was the foundation behind all modern cluster technologies. Kubernetes, Nomad, Mesos, Rancher and Swarm are all implementations of Borg. A key design feature of Borg is that already running tasks continue to run even if the Borg master, or other tasks (a.k.a. Borglets) go down. Machine nodes that are part of a cluster are designated as either a client or server. Clients are optimized for availability and run tasks assigned by servers. Servers are responsible for managing configuration state and are optimized for consistency in the event of an outage. There are two kinds of servers: leaders and followers and only one server is allowed to be the leader at a time.

Cluster federation is an extension of the normal clustering idea and occurs when two or more clusters network themselves together over WAN (or more likely, private VPN) to form a loose coupling between otherwise isolated datacenters. This virtual bridge that is formed means that all of your compute capacity can be managed from a single control plane. Cluster federation is indispensable for organizations wishing to support operations in multiple clouds, without needing proportionally large teams of engineers to do so. Federation is not any harder to do than normal clustering, although special attention does need to be paid to Access Control Lists (ACLs) and firewall rules. This is not important for now, since I've already configured everything in advance, but is something to keep in mind when federating your own clusters.

### 8.2.2 Architecture

You can think of this project as having three distinct groups of virtual machines, one to run the Consul server, one to run the Nomad server, and one to run the Nomad client. All nodes register themselves with Consul to enable service-to-service discovery, so there is effectively one large Consul cluster, while the Nomad cluster is a subset of this, consisting only of the Nomad servers and Nomad clients. A conceptual architecture diagram of these three groups is shown in figure 8.8

**NOTE** Consul and Nomad follow raft consensus protocol, meaning there must be an odd number of servers (with a minimum number of three) to have quorum. One of these servers is designated the leader while the others are followers. There is no such restrictions for clients.

<sup>5</sup> <https://ai.google/research/pubs/pub43438>

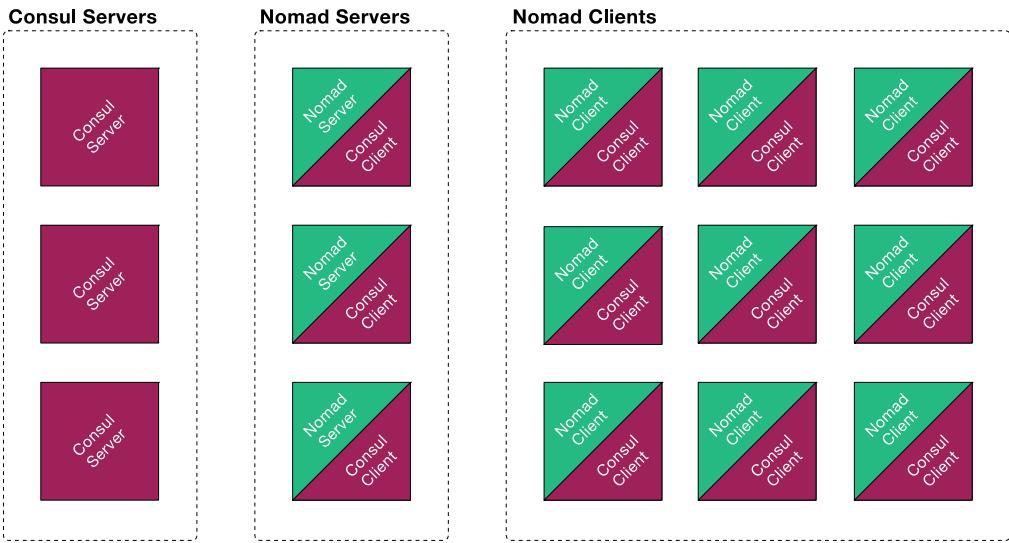


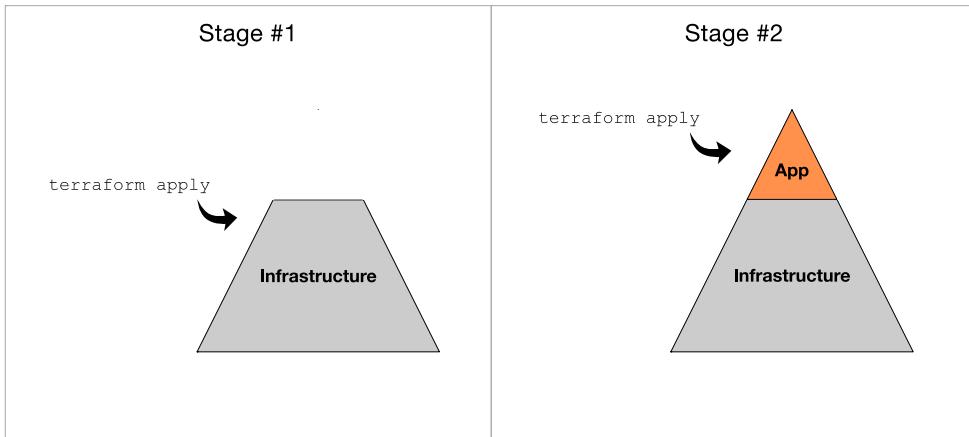
Figure 8.8 there are three distinct groups of virtual machines, one to run the Consul server, one to run the Nomad server, and one to run the Nomad client. Effectively, there is one large Consul cluster, with a subset comprising the Nomad cluster.

These three groups of virtual machines are replicated in both clouds, and like-to-like clusters are federated together. A detailed architecture diagram is shown in figure 8.9. Notice that the number of Nomad clients have been limited to three, primarily due to Azure restricting the number of CPU cores that are allowed to be launched in a free tier account (without putting in a support ticket request).



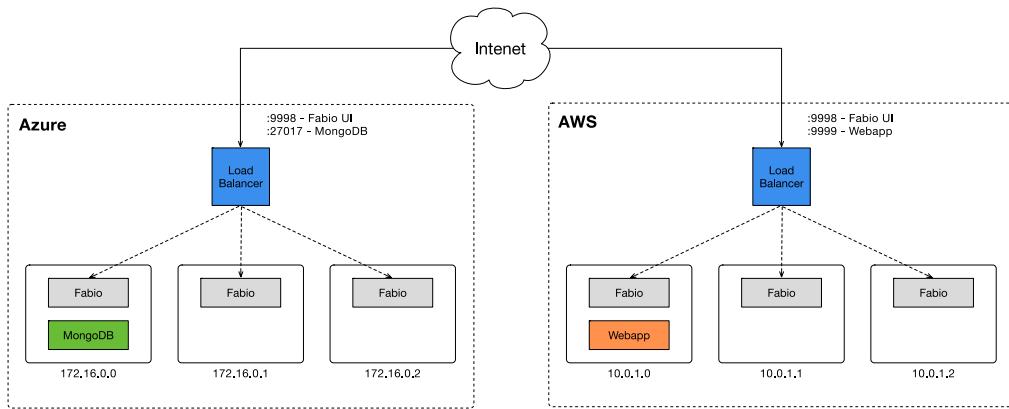
Figure 8.9 Detailed architecture diagram of how federation occurs between the Consul servers and Nomad servers, respectively. The little gold crowns are because among the machines running in “server” mode, one is designated as a leader while the others are followers (following raft protocol)

As far as implementation goes, we will be performing a two-stage deployment, similar to what we did in chapter 7. The first stage provisions base level infrastructure and bootstraps the clusters, while the second stage deploys application containers on top of that. A slight deviation from what we did in chapter 7 is that both stages will be deployed with Terraform (rather than one with Terraform and one with Git). Figure 8.10 illustrates the two-stage deployment process.



**Figure 8.10 Deployment is done in two stages. First the core infrastructure is provisioned, then the infrastructure comprising the application is deployed on top of that**

A detailed network topology for the application layer (stage two) is depicted in Figure 8.11. The application layer is composed of two docker containers, one for the webapp and one for the database. While the frontend webapp runs on AWS, the Mongo database runs on Azure. Each Nomad cluster runs Fabio<sup>6</sup> on all clients for application load balancing/routing. Fabio is exposed to the outside world through an external network load balancer which was deployed as part of stage one.



**Figure 8.11 Network topology for application layer. MongoDB runs on Azure, the MMORPG app runs in AWS, and**

<sup>6</sup> Fabio is an HTTP and TCP reverse proxy that configures itself with data from Consul <https://fabiolb.net/>

Fabio runs on all Nomad clients for application load balancing.

### 8.2.3 Stage #1 Base Infrastructure

Now that we have the background and architecture discussion out of the way, let's start writing the infrastructure code for stage one. Create a `variables.tf` to configure the AWS and Azure providers.

#### **Listing 8.7 variables.tf**

```
variable "aws" {
  type = object({
    profile = string
    region = string
  })
}

variable "azure" {
  type = object({
    subscription_id = string
    client_id       = string
    client_secret   = string
    tenant_id       = string
    location        = string
  })
}
```

As before, we need to set the values with a `terraform.tfvars` file. An example is shown in Listing 8.8.

#### **Listing 8.8 terraform.tfvars**

```
aws = {
  profile = "<profile>"
  region = "us-west-2"
}

azure = {
  subscription_id = "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxx"
  client_id       = "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxx"
  client_secret   = "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxx"
  tenant_id       = "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxx"
  location        = "centralus"
}
```

These variables will be used to configure the AWS and Azure Terraform providers:

#### **Listing 8.9 providers.tf**

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

```

}

provider "azurerm" {
  subscription_id = var.azure.subscription_id
  client_id      = var.azure.client_id
  client_secret   = var.azure.client_secret
  tenant_id      = var.azure.tenant_id
}

```

There are two modules that deploy all the project resources. I regret that I do not have space to go into depth about how these modules work, but I encourage you to look at the source code in my public GitHub repo if you wish to learn more. These modules deploy the VPCs, load balancers, autoscaling groups, security groups, and so on to start up and federate Consul and Nomad clusters on AWS and Azure. This code can easily be modified to launch arbitrary clusters (such as Vault or Kubernetes). If nothing else, you may find it useful to see how the same problem is solved on Azure, as compared with AWS.

Listing 8.10 has the code for configuring the modules. In this code, you have the ability to select what version of Consul/Nomad you want, how many nodes per cluster, and what size machines you want to use. The Azure module also accepts a `join_wan` attribute which takes the IP addresses from the AWS module to federate the Consul clusters. You don't need to change anything; the defaults are fine.

#### **Listing 8.10 main.tf**

```

module "aws" {
  source      = "scottwinkler/nomad/aws"
  associate_public_ips = true #A

  consul = { #B
    version      = "1.6.1"
    servers_count = 3
    server_instance_type = "t3.micro"
  }

  nomad = { #C
    version      = "0.9.5"
    servers_count = 3
    server_instance_type = "t3.micro"
    clients_count     = 3
    client_instance_type = "t3.micro"
  }
}

module "azure" {
  source      = "scottwinkler/nomad/azure"
  azure       = var.azure
  associate_public_ips = true #A
  join_wan    = module.aws.public_ips.consul_servers #D

  consul = { #B
    version      = "1.6.1"
  }
}

```

```

servers_count      = 3
server_instance_size = "Standard_A1"
}

nomad = { #C
  version        = "0.9.5"
  servers_count   = 3
  server_instance_size = "Standard_A1"
  clients_count    = 3
  client_instance_size = "Standard_A1"
}
}

```

#A because we do not have a VPN tunnel setup between Azure and AWS, we have to associate public IP addresses and use for joining the clusters together.

#B configuration block for the Consul cluster

#C configuration block for the Nomad cluster

#D The Azure Consul cluster federates itself with the AWS Consul cluster using the public IP addresses from the AWS module

For convenience, we will bubble up each of the modules into an output value. This gives us information that we will need in the next section.

#### **Listing 8.11 outputs.tf**

```

output "aws" {
  value = module.aws #A
}

output "azure" {
  value = module.azure #A
}

```

#A outputting all the outputs of a module is a great Terraform 0.12 feature

Finally, create a `versions.tf` as laid out in Listing 8.12.

#### **Listing 8.12 versions.tf**

```

terraform {
  required_version = "~> 0.12"
  required_providers {
    aws   = "~> 2.29"
    azurerm = "~> 1.34"
    random = "~> 2.2"
  }
}

```

### **8.2.4 Deploying Base Infrastructure**

Let's now deploy the base infrastructure. Initialize the workspace with a `terraform init` and run a `terraform apply` to get started.

```
$ terraform apply
...
+ resource "aws_iam_role_policy" "iam_role_policy" {
  + id   = (known after apply)
  + name = (known after apply)
  + policy = jsonencode(
    {
      + Statement = [
        + {
          + Action  = [
            + "logs:*",
            + "ec2:DescribeInstances",
          ]
          + Effect  = "Allow"
          + Resource = "*"
          + Sid     = ""
        },
      ],
      + Version  = "2012-10-17"
    }
  )
  + role  = (known after apply)
}
```

Plan: 93 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?

Terraform will perform the actions described above.  
Only 'yes' will be accepted to approve.

Enter a value:

After approving the apply and waiting approximately 10-15 minutes for all the infrastructure to be provisioned, the tail end of the output will be something like:

```
...
Apply complete! Resources: 93 added, 0 changed, 0 destroyed.
```

Outputs:

```
aws = {
  "addresses" = {
    "consul_ui" = "http://terraformaction-7p7ma0-consul-1023640368.us-west-2.elb.amazonaws.com:8500"
    "fabio_lb" = "http://terraformaction-7p7ma0-fabio-6ad80758f451730b.elb.us-west-2.amazonaws.com:9999"
    "fabio_ui" = "http://terraformaction-7p7ma0-fabio-6ad80758f451730b.elb.us-west-2.amazonaws.com:9998"
    "nomad_ui" = "http://terraformaction-7p7ma0-nomad-398143868.us-west-2.elb.amazonaws.com:4646"
  }
  "public_ips" = {
    "consul_servers" = [
      "34.222.121.229",
      "34.217.39.12",
      "18.237.241.247",
    ]
    "nomad_servers" = [
      "34.216.39.34",
    ]
  }
}
```

```

    "34.210.13.254",
    "34.222.93.245",
]
}
}
azure = {
"addresses" = {
  "consul_ui" = "http://terraforminaction-t2ndbv-consul.centralus.cloudapp.azure.com:8500"
  "fabio_db" = "tcp://terraforminaction-t2ndbv-fabio.centralus.cloudapp.azure.com:27017"
  "fabio_ui" = "http://terraforminaction-t2ndbv-fabio.centralus.cloudapp.azure.com:9998"
  "nomad_ui" = "http://terraforminaction-t2ndbv-nomad.centralus.cloudapp.azure.com:4646"
}
}

```

**WARNING!** Exposing Consul and Nomad without ACLs and over plain HTTP is not okay for production deployments. If you are interested in using these modules as templates for production use, I would recommend forking the GitHub repo(s) and adding this configuration in yourself.

We can check with the Consul UI (which was launched automatically during `terraform apply`) to verify the base level services are registering themselves with Consul, and check if federation has occurred between the Consul clusters. Copy the URL from either `aws.addresses.consul_ui` or `azure.addresses.consul_ui` and paste in into the browser. You will get something that looks like:

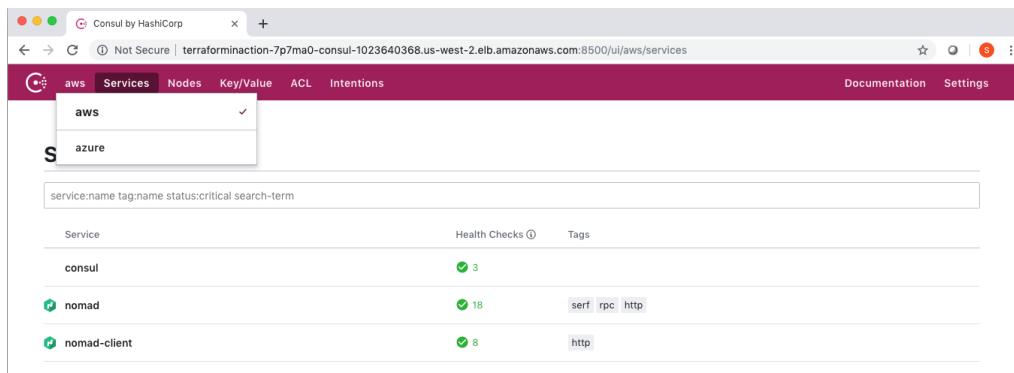


Figure 8.12 The AWS Consul has started up, federated with the Azure cluster, and both the Nomad servers and clients have registered themselves with Consul. Clicking on the services tab will allow you to toggle between the AWS and Azure Consul clusters.

**NOTE** although Terraform has applied successfully, it will still take a few minutes for the clusters to finish initializing. In the meantime, you may get weird error messages while the servers are establishing quorum.

Once Consul is up, you can check Nomad to verify that all the Nomad clients have registered themselves with their respective Nomad servers. Follow the URL for either

`aws.addresses.nomad_ui` or `azure.addresses.nomad_ui` and click on the tab for "clients". You should get something like figure 8.13:

The screenshot shows the Nomad UI interface with the title bar "Clients - centralus - Nomad". Below the title bar is a navigation bar with links for "Documentation" and "ACL Tokens". The main area is titled "Clients" and contains a search bar with placeholder text "Search clients..." and a clear button "X". To the right of the search bar are three dropdown filters: "Class", "State", and "Datacenter". A table below the filters lists three clients with the following details:

ID	Name	State	Address	Datacenter	# Allocs
ccbd3592	terraformaction-fu5...	ready	172.16.0.13:4646	azure	0
338771a5	terraformaction-fu5...	ready	172.16.0.11:4646	azure	0
9a9811b7	terraformaction-fu5...	ready	172.16.0.12:4646	azure	0

At the bottom right of the table, there is a page number "1-3 of 3".

Figure 8.13 Nomad clients have joined the cluster and are ready to work. In the top left, if you click on the regions tab, you can switch to the AWS view.

### 8.2.5 Stage #2 Application Infrastructure

We are now ready for stage two: deploying the application containers on top of the base infrastructure. We will use the Nomad provider for Terraform to accomplish this, although in real-world situations you would likely be better off using either the SDK, CLI or raw API as part of an automated CI/CD pipeline (the Nomad provider suffers many of the same problems the Docker provider has). The reason we are covering the Terraform way is because there is still value in understanding how the Nomad provider works and because we will soon be writing our own provider in chapter 10.

Create a new Terraform workspace with a single file called `nomad.tf` containing the contents presented in Listing 8.13. You will need to populate it with some of the addresses from the previous section.

#### Listing 8.13 nomad.tf

```
terraform {
  required_version = "~> 0.12"
  required_providers {
    nomad   = "~> 1.4"
```

```

}

provider "nomad" { #A
  address = "<aws.addresses.nomad_ui>"
  alias   = "aws"
}

provider "nomad" { #A
  address = "<azure.addresses.nomad_ui>"
  alias   = "azure"
}

module "mmorpg" {
  source  = "scottwinkler/mmorpg/nomad"
  fabio_db = "<azure.addresses.fabio_db>" #B
  fabio_lb = "<aws.addresses.fabio_lb>" #B
  providers = { #C
    nomad.aws  = "nomad.aws"
    nomad.azure = "nomad.azure"
  }
}

output "browserquest_address" {
  value = module.mmorpg.browserquest_address
}

```

#A The Nomad provider needs to be declared twice because of an oddity with how the API handles jobs  
#B The module needs to know the address of the database and load balancer to initialize. Consul could be used for service discovery, but that would require the two clouds to have a private network tunnel to each other  
#C This meta-attribute is a reserved keyword to allow the providers to be explicitly passed into the module

Next, initialize Terraform and run an apply.

```
$ terraform apply
...
+ task_groups      = [
  + {
    + count = 1
    + meta  = (known after apply)#A
    + name  = "mongo"
    + task  = [
      + {
        + driver = "docker"
        + meta   = (known after apply)
        + name   = "server"
      },
    ],
  ],
]
+ type             = "service"
}
```

Plan: 4 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?

Terraform will perform the actions described above.  
Only 'yes' will be accepted to approve.

Enter a value:

#A "known after apply" just means it is a computed attribute

Confirm the apply and the containers will be deployed onto Nomad.

```
...
module.mmmrpg.nomad_job.azure_fabio: Creation complete after 0s [id=fabio]
module.mmmrpg.nomad_job.aws_fabio: Creation complete after 0s [id=fabio]
module.mmmrpg.nomad_job.aws_browserquest: Creation complete after 0s [id=Browserquest]
```

Apply complete! Resources: 4 added, 0 changed, 0 destroyed.

Outputs:

```
browserquest_address = http://terraformaction-7p7ma0-fabio-6ad80758f451730b.elb.us-west-2.amazonaws.com:9999
```

You can inspect the health of the application by checking with Nomad, Consul, and Fabio in that order. Figures 8.13-8.16 illustrate what this process looks like.

The screenshot shows the Nomad UI interface. At the top, there's a header bar with tabs for 'Jobs - us-west-2 - Nomad'. Below the header, a search bar says 'Search jobs...'. To the right of the search bar are filters for 'Type', 'Status', 'Datacenter', and 'Prefix', followed by a 'Run Job' button. On the left, there's a sidebar with tabs for 'WORKLOAD' (selected), 'Jobs' (selected), 'CLUSTER', 'Clients', and 'Servers'. The main content area displays a table of jobs. The table has columns for 'Name', 'Status', 'Type', 'Priority', 'Groups', and 'Summary'. There are two entries: 'browserquest' (Status: RUNNING, Type: service, Priority: 50, Groups: 1) and 'fabio' (Status: RUNNING, Type: system, Priority: 50, Groups: 1). A footer at the bottom of the table says '1-2 of 2'.

Name	Status	Type	Priority	Groups	Summary
browserquest	RUNNING	service	50	1	<div style="width: 100%; background-color: #2e8b57; height: 10px;"></div>
fabio	RUNNING	system	50	1	<div style="width: 100%; background-color: #2e8b57; height: 10px;"></div>

Figure 8.14 in the Nomad UI you can see that the BrowserQuest and Fabio jobs are currently running in the AWS region. You can click on the regions tab to you can switch over to the Azure view and see Fabio and MongoDB running there.

The screenshot shows the Consul UI interface. At the top, there's a navigation bar with tabs for 'aws', 'Services', 'Nodes', 'Key/Value', 'ACL', and 'Intentions'. On the right side of the header are 'Documentation' and 'Settings' links. Below the header, the main content area has a title 'Services 5 total'. A search bar at the top of this section contains the placeholder 'service:name tag:name status:critical search-term'. Below the search bar is a table with five rows, each representing a service. The columns are 'Service', 'Health Checks', and 'Tags'. The services listed are:

Service	Health Checks	Tags
browserquest-aws	2	urlprefix/-
consul	3	
fabio	6	
nomad	18	serf rpc http
nomad-client	6	http

**Figure 8.15** Since the jobs are running, they will register themselves as services with Consul, which can be seen in the Consul UI

The screenshot shows the Fabio UI interface. The top navigation bar includes a back/forward button, a refresh icon, and the URL 'Not Secure | terraforminaction-7p7ma0-fabio-6ad80758f451730b.elb.us-west-2.amazonaws.com:9998/routes'. The main title is 'fabio'. On the right, there are 'Overrides', '1.5.11', and 'Github' buttons. Below the title, the heading 'Routing Table' is displayed, followed by a search bar with the placeholder 'type to filter routes'. A table below lists the routing rules. At the bottom center is the 'Fabio' logo.

#	Service	Source	Dest	Options	Weight
1	browserquest-aws	/	http://10.0.103.19:31090/	http://10.0.103.19:31090/	100.00%

**Figure 8.16** After the services are marked as healthy by Consul, they are able to be detected by Fabio. In AWS, Fabio routes HTTP traffic to the dynamic port that BrowserQuest is running on, while in Azure, Fabio routes TCP traffic to the dynamic port MongoDB is running on.

## 8.2.6 Ready Player One

After verifying the health of the service, you are ready to play! Copy the `browserquest_address` output into your browser you will be presented with a screen asking to create a new character. Anyone who has this address (e.g. other clients) can join the game and play with you.

**NOTE** The title screen says “Phaser Quest” instead of “Browser Quest” because it is a faithful recreation of the original game using the Phaser game engine for Javascript. Credit goes to Jerenaux<sup>7</sup>

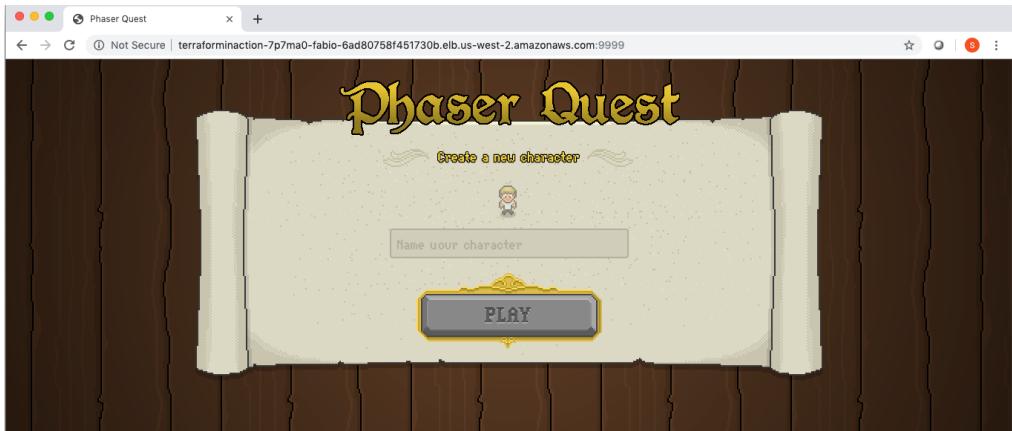


Figure 8.17 welcome screen for the BrowserQuest MMORPG. You are now able to create a character and anyone who has the link can play with you

When you are done, remember to tear down the base level infrastructure before proceeding. After all, you don't want to be paying for infrastructure you aren't even using.

```
$ terraform destroy -auto-approve
...
module.azure.module.resourcegroup.azure_rm_resource_group: Destruction complete after 46s
module.azure.module.resourcegroup.random_string.rand: Destroying... [id=t2ndbvgi4ayw2qmhl7mw1bu]
module.azure.module.resourcegroup.random_string.rand: Destruction complete after 0s

Destroy complete! Resources: 93 destroyed.
```

### 8.3 Rearchitecting the MMORPG to use Managed Services

Think of this as a bonus section. I could have ended with the previous scenario, but I feel the overall story would have been incomplete. The magical thing about multi-cloud is that it is whatever you want it to be. Multi-cloud doesn't have to mean replicating services horizontally for fault tolerance, or federating container orchestration platforms for standardizing compute capacity. Another perfectly valid way to multi-cloud is to mix and match managed service offerings in whatever way you see fit.

<sup>7</sup> <https://www.github.com/Jerenaux/phaserquest>

By “managed services”, I mean anything that isn’t raw compute or heavy on the operations side of things; both SaaS and Serverless qualify under this definition. Managed services are convenient and are also what make each cloud unique. Even the same type of managed service differs in implementation across clouds – these differences can be perceived either as an obstacle or an opportunity. I prefer the latter.

In this section we will be rearchitecting the MMORPG to run on managed services in AWS and Azure. More specifically, we will be using AWS Fargate to deploy a serverless container, and Azure CosmosDB to deploy a managed MongoDB. The Fargate container runs the MMORPG application and connects to the Azure database over WAN using the primary connection string. A simplified architecture diagram is depicted in figure 8.18.

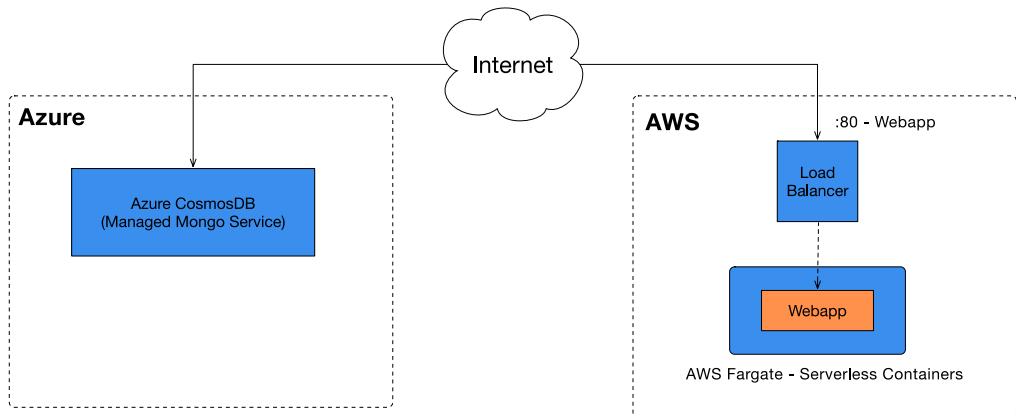


Figure 8.18 Architecture for multi-cloud deployment of MMORPG using managed services

### Lego Bricks Metaphor

In many ways, working with Terraform is a lot like building with Lego bricks. Terraform has all these providers, which much like individual Lego sets, give you a huge assortment of pieces to work with. You don't need any specialized tools to assemble them – they just fit together because that's how they were designed.

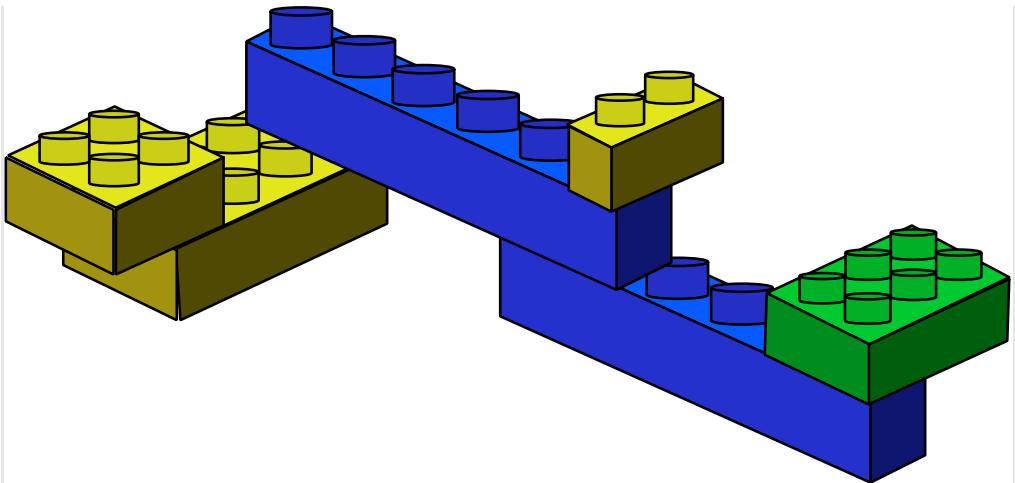


Figure 8.19 Combining resources from various cloud providers is like building with Lego blocks

Naturally, building something new is always the hard part. Sometimes, as with Legos, you have more blocks than you know what to do with, or some of the blocks you need are missing entirely (and you don't even know which those are). Also, the instructions may be missing, or partially missing, but you still have to build that Millennium Falcon anyways. Given the sheer number of blocks at hand, it's inevitable that there will be good and bad ways to combine blocks together.

I am not proposing that it is always a good idea to mix and match resources between various cloud providers – that would be silly. Instead, I aim to encourage you to keep an open mind when working with Terraform. The best design may not always be the most obvious one.

### 8.3.1 Code

This has already been a long chapter, so I will make this quick. There is just one file you need to create, and it has everything needed to deploy this scenario. Create a new workspace with a `player2.tf` file, including the contents from Listing 8.14.

#### **Listing 8.14 player2.tf**

```
terraform {
  required_version = "~> 0.12"
  required_providers {
    aws  = "~> 2.29"
    azurerm = "~> 1.34"
    random = "~> 2.2"
  }
}

provider "aws" {
  profile = "<profile>"
  region = "us-west-2"
```

```

}

provider "azurerm" {
  subscription_id = "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxx"
  client_id      = "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxx"
  client_secret   = "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxx"
  tenant_id      = "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxx"
}

module "aws" {
  source = "scottwinkler/mmorpg/cloud//aws"
  app = {
    image  = "swinkler/browserquest"
    port   = 8080
    command = "node server.js --connectionString ${module.azure.connection_string}"
  }
}

module "azure" {
  source  = "scottwinkler/mmorpg/cloud//azure"
  namespace = "terraforminaction"
  location = "centralus"
}

output "browserquest_address" {
  value = module.aws.lb_dns_name
}

```

### 8.3.2 Ready Player Two

We are ready to deploy! Easy, right? Initialize the workspace with a `terraform init` followed by a `terraform apply`. The result of the `terraform apply` is:

```
$ terraform apply
...
+ name          = (known after apply)
+ owner_id      = (known after apply)
+ revoke_rules_on_delete = false
+ vpc_id        = (known after apply)
}
```

Plan: 37 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?  
Terraform will perform the actions described above.  
Only 'yes' will be accepted to approve.

Enter a value:

Confirm the apply and wait until Terraform does its thing.

```
...
module.aws.aws_ecs_task_definition.ecs_task_definition: Creation complete after 1s [id=terraforminaction-ebfes6-app]
module.aws.aws_ecs_service.ecs_service: Creating...
module.aws.aws_ecs_service.ecs_service: Creation complete after 0s [id=arn:aws:ecs:us-west-
```

```
2:215974853022:service/terraformaction-ebfes6-ecs-service]

Apply complete! Resources: 37 added, 0 changed, 0 destroyed.

Outputs:

browserquest_address = terraformaction-ebfes6-lb-444442925.us-west-2.elb.amazonaws.com
```

Now copy the BrowserQuest address into the browser and you are ready to play! Be patient though, because it can take a few minutes before the 503 error goes away.



**Figure 8.20 Multi-cloud means multiplayer!**

**TIP** remember to tear down the infrastructure with a `terraform destroy` when you are done to avoid paying additional costs!

## 8.4 Fireside Chat

We started this chapter by deploying a hybrid cloud load balancer to demonstrate how easy it is to multi-cloud with Terraform. Multi-cloud doesn't have to mean crazy complex; it can be done on the quick and dirty if you know what you want and design for that. Nevertheless, multi-cloud doesn't guarantee you any more resiliency and redundancy than you could get with a single cloud. You still need to put on your architect hat if you want that.

The next example we deployed used a two-stage deployment process to launch a container orchestration platform and then run BrowserQuest. This technique can also be applied to provisioning and configuring other container orchestration platforms -such as Kubernetes or Docker Swarm, as well as more general clusters (such as Istio, Kafka, Zookeeper, etc.).

We didn't stop at deploying one cluster however, we deployed two clusters and federated them together over WAN. Federation provided us a single control plane from which to manage all our compute capacity and is one of the most popular strategies for hybrid/multi-cloud architectures.

Because of latency between datacenters, you probably would not run your database in a separate region than your web application, but at least you know that it is possible!

Lastly, we followed up by rearchitecting the MMORPG to run on managed services. This tech comparison poignantly illustrates that you don't have to go all in on Kubernetes or Nomad if you don't want to. Being able to pick and choose best-in-class services is often more beneficial than managing your own infrastructure. Terraform is the glue that binds managed services together as part of a single deployment; you don't need fancy clusters or container orchestration platforms for that.

## 8.5 Summary

- Terraform can be used to orchestrate hybrid and multi-cloud deployments. It is no harder than using Terraform for the single cloud.
- Not all Terraform providers are worthwhile. For example, the Nomad and Docker providers for Terraform offer questionable value at best. It may be easier to call the API directly than to incorporate these into your workflows.
- Cluster federation can be done automatically as part of a `terraform apply`, although it won't necessarily be ready when Terraform returns. This is called "bootstrapping".
- Two-stage deployments are best when there is a clear distinction between "infrastructure" and "application". Infrastructure is any of the virtual machines, clusters and managed services that applications run on. Each stage should have its own state.
- Terraform can be used to deploy containerized services, whether that be in the traditional sense – via container orchestration platforms – or in the more modern sense – via a mix of managed service offerings.
- Multi-cloud can be implemented however you like, there is no "right" way to do it.

# 9

## Zero Downtime Deployments

### This chapter covers:

- Customizing resource lifecycles with the `create_before_destroy` flag
- Performing Blue/Green deployments with Terraform
- Combining Terraform with Ansible
- Generating SSH keypairs automatically with the TLS provider
- Installing software on virtual machines with remote-exec provisioners

Traditionally, there has been a window of time during software deployments when servers are incapable of serving production traffic. This window is typically scheduled for early morning off-hours to minimize downtime, but it still impacts availability. *Zero Downtime Deployment* (ZDD) is the practice of keeping services always running and available to customers, even in the midst of software updates. If executed well, users should not even be aware when changes are being made to the system.

In this chapter we will investigate three approaches to achieving zero downtime deployments with Terraform. First, we will use the `create_before_destroy` meta-attribute to ensure that an application is running and passing health checks before tearing down the old one. The `create_before_destroy` meta-attribute alters how *force new* updates are internally handled by Terraform. By setting it to `true`, interesting and unexpected behavior can result.

Next, we will examine one of the oldest and most popular ways to achieve ZDD: *Blue/Green deployments*. This technique makes use of two separate environments (one “Blue” and the other “Green”) to rapidly cutover from one software version to another. Blue/Green is popular because it is fairly easy to implement and enables rapid rollback. Furthermore, Blue/Green is a stepping-stone towards more advanced forms of ZDD, such as rolling Blue/Green and canary deployments.

Lastly, we will offload the responsibilities of ZDD to another, more suitable technology. Specifically, Ansible. Ansible is a popular configuration management tool that allows you to

rapidly deploy applications onto existing infrastructure. By provisioning all your static infrastructure with Terraform, Ansible can be left to deploy the more dynamic applications.

## 9.1 Lifecycle Customizations

Consider a resource that provisions an instance in AWS which starts a simple HTTP server running on port 80:

### Snippet 9.1 EC2 Instance

```
resource "aws_instance" "instance" {
  ami           = var.ami
  instance_type = var.instance_type

  user_data = <<-EOF #A
    #!/bin/bash
    mkdir -p /var/www && cd /var/www
    echo "App v${var.version}" >> index.html
    python3 -m http.server 80
  EOF
}
```

#A starts a simple HTTP webserver

If one of the force new attributes (`ami`, `instance_type`, `user_data`) were to be modified, then during a subsequent `terraform apply`, the existing resource would be destroyed before the new one is created. This is the default behavior for Terraform. The drawback is that there is downtime between when the old resource has been destroyed and the replacement resource has yet to be provisioned. This downtime is not negligible and can be anywhere from 5 minutes, to an hour, or more, depending on the upstream API.

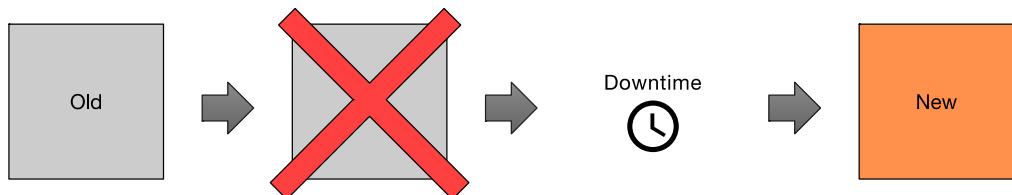


Figure 9.1 By default any `force new` update on a resource results in downtime. This is because the old resource must first be destroyed before a new resource can be created.

To avoid downtime, there is a `lifecycle` meta-argument that allows you to customize the resource lifecycle. The `lifecycle` nested block is present on all resources and allows you to set the following three flags:

- `create_before_destroy` (bool) – when set to `true`, the replacement object is created before the old object is destroyed
- `prevent_destroy` (bool) – when set to `true`, Terraform will reject any plan that would destroy the infrastructure object associated with the resource with an explicit error
- `ignore_changes` (list of attribute names) – specifies a list of resource attributes that Terraform should ignore when generating execution plans. This allows a resource to have some measure of configuration drift, without forcing updates to occur.

These three flags allow you to override default behavior for resource creation, destruction, and updates, and should be used with extreme caution because they alter the fundamental behavior of Terraform.

### 9.1.1 Zero Downtime Deployments with `create_before_destroy`

The most intriguing parameter on the `lifecycle` block is `create_before_destroy`. This flag switches the ordering in which Terraform performs a force new update. When set to `true`, the new resource is provisioned alongside the existing resource. Only after the new resource has successfully been created is the old resource destroyed. This concept is shown in figure 9.2.

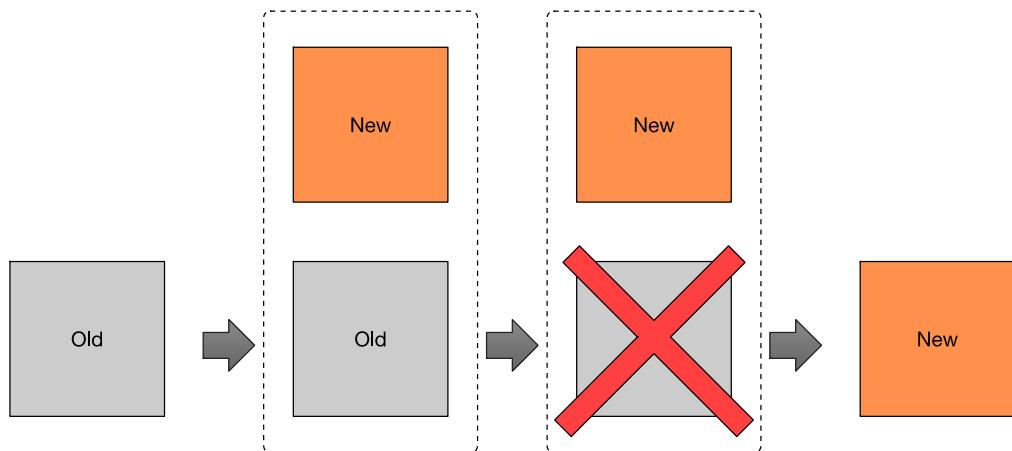


Figure 9.2 when `create_before_destroy` is set to `true`, the replacement resource is created before the old resource is destroyed. This means you don't experience any downtime during force new updates.

**NOTE** `create_before_destroy` doesn't default to `true` because many providers do not allow two instances of the same resource to exist at once. For example, you can't have two S3 buckets with the same name.

Paul Hinzie, Director of Engineering at HashiCorp, suggested back in 2015<sup>4</sup> that the `create_before_destroy` flag could be used to enable Zero Downtime Deployments (ZDD). Consider code snippet 9.2, which modifies the lifecycle of an `aws_instance` resource by setting the `create_before_destroy` flag to `true`.

### Snippet 9.2 EC2 Instance with `create_before_destroy`

```
resource "aws_instance" "instance" {
  ami           = var.ami
  instance_type = "t3.micro"

  lifecycle {
    create_before_destroy = true
  }

  user_data = <<-EOF
  #!/bin/bash
  mkdir -p /var/www && cd /var/www
  echo "App v${var.version}" >> index.html
  python3 -m http.server 80
  EOF
}
```

As before, any changes to one of the force new attributes will trigger a force new update, but because `create_before_destroy` is now set to `true`, the replacement resource will be created before the old one is destroyed. This applies only to managed resource (i.e. not data sources).

Suppose `var.version`, a variable denoting the application version, were incremented from "1.0" to "2.0". This would trigger a force new update on `aws_instance`, because it alters `user_data`, which is a force new attribute. Even with `create_before_destroy` set to `true`, however, we cannot guarantee that the HTTP server will be running after the resource has been marked as "created". In fact, more than likely, it won't be. This is because Terraform manages things that it knows about (the EC2 instance) and not the application that runs on that instance (the HTTP server).

We can circumvent this limitation by taking advantage of resource provisioners. Due to the way provisioners were implemented, a resource is not marked as "created" or "destroyed" unless all creation-time and destruction-time provisioners have executed with no errors. This means that we can use a local-exec provisioner to perform creation-time health checks to ensure that not only has the instance been created, but that the application is healthy and serving HTTP traffic (see Snippet 9.3).

### Snippet 9.3 EC2 Instance with Health Check

```
resource "aws_instance" "instance" {
  ami           = var.ami
  instance_type = "t3.micro"

  lifecycle {
    create_before_destroy = true
  }
```

---

<sup>4</sup> <https://groups.google.com/forum/#!msg/terraform-tool/7Gdhv10Ac80/INQ93riLwAJ>

```

}

user_data = <<-EOF
#!/bin/bash
mkdir -p /var/www && cd /var/www
echo "App v${var.version}" >> index.html
python3 -m http.server 80
EOF

provisioner "local-exec" { #A
  command = "./health-check.sh ${self.public_ip}"
}
}

```

#A application health check. The script file health-check.sh is presumed to exist

**NOTE** the `self` object within a local-exec provisioner is a reference to the current resource the provisioner is attached to

### 9.1.2 Additional Considerations

Although it would appear that `create_before_destroy` is an easy way to perform zero downtime deployments, it does have a number of quirks and shortcomings that should be kept in mind:

- **Confusing** – once you start messing with the default behavior of Terraform, it's harder to reason about how changes to your configuration files and variables will affect the outcome of an apply. This is especially true when `local-exec` provisioners are thrown in the mix.
- **Redundant** – Everything you could accomplish with `create_before_destroy` could also be done with two Terraform workspaces or modules
- **Namespace Collisions** – because both the new and old resource must exist at the same time, you have to choose parameters that will not conflict with each other. This is often awkward, and sometimes even impossible, depending on how the parent provider implemented the resource.
- **Force New vs. In-place** – not all attributes force the creation of a new resource. Some attributes (like tags on AWS resources) are updated in-place, which means that the old resource is never actually destroyed, but merely altered. This also means any attached resource provisioners won't be triggered.

**TIP** I personally do not use `create_before_destroy` as I have found it to be more trouble than it is worth

## 9.2 Blue/Green Deployments

Blue/Green deployments are when you have two production environments that you switch between, one called “Blue” and one called “Green”. Only one production environment is live at any given time. A router directs traffic to the live environment and can be either a load balancer or a DNS resolver. Whenever you want to deploy to production, you first deploy to the idle environment. Then, when you are ready, you switch the router to point from the live server

over to the idle server – which is already running the latest version of the software. This switch is referred to as *cutover* and can be done manually or automatically. When the cutover completes, the idle server becomes the new live server and the former live server is now the idle server. See figure 9.3 for a visual overview.

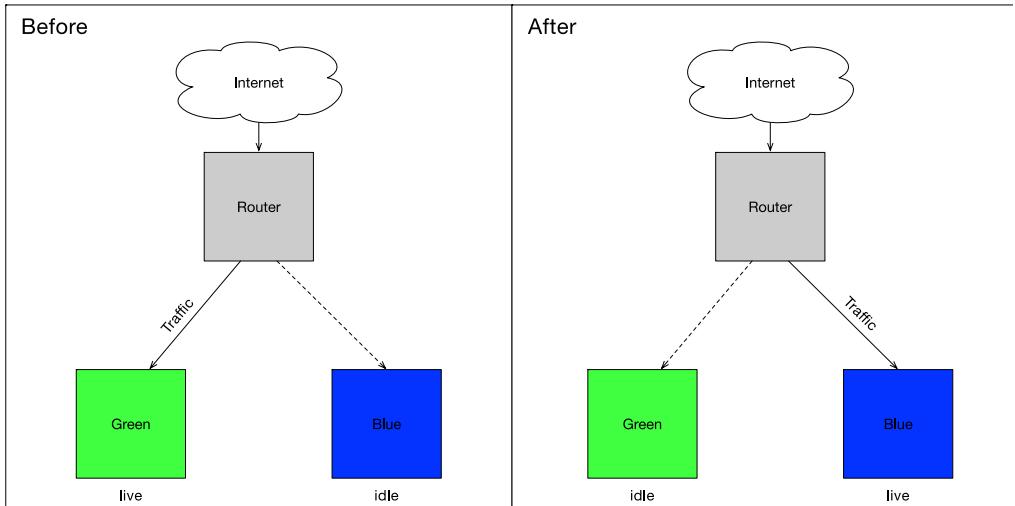


Figure 9.3 Blue/Green deployments have two production environments: one live and serving production traffic and the other idle. Changes are always made first to the idle environment and then a cutover is made from live to idle.

### Video Graphics Analogy

Suppose you had to draw an entire picture on the screen, pixel-by-pixel or line-by-line. If you were to draw such an image to the screen directly, you would immediately be disappointed by how long it takes. This is because there is a hard limit for how fast changes can be propagated to the screen – usually between 60 and 120Hz (or cycles per seconds). Most programmers use a technique called *double-buffering* to combat this problem. Double-buffering is the act of writing to an in-memory data structure called a “back buffer”, then drawing the image from the back buffer to the screen in a single operation. This technique is significantly faster than drawing pixels one at a time and is good enough for most applications.

However, for some particularly graphics intensive applications – namely, video games – double-buffering is still too slow. This is because there is still downtime, as the graphics card cannot be writing to the back buffer at the same time the screen is reading from it (and vice versa). A clever workaround is to utilize not one but two back buffers. One back buffer is reserved for the screen while the other is reserved for the graphics card. After some pre-defined period of time, the back buffers are swapped (i.e. the screen pointer is cutover from one back buffer to the other). This technique, called *page flipping*, is a fun analogy to how Blue/Green deployment works.

Blue/Green deployments are the oldest and most popular way to achieve zero downtime deployments. More advanced implementation of ZDD would be something like Rolling Blue/Green<sup>2</sup> and/or Canary deployments<sup>3</sup>.

### **Rolling Blue/Green and Canary Deployments**

Rolling Blue/Green is similar to regular Blue/Green, except instead of moving 100% of the traffic over at once, you slowly replace one server at a time. Suppose you have a set of production servers running the old version of your software and wish to update to the new version. To commence rolling Blue/Green, you would launch a new server running the new version, ensure it passes health checks, and then kill one of the old servers. You would do this incrementally, one at a time, until all servers are migrated over and running the latest version of the software. This is more complicated from an application standpoint as you have to ensure the application can support running two versions concurrently. Some stateful applications may have trouble with data corruption if the schema changes from one version to the next while read/writes are still taking place.

Canary deployments are also about deploying an application in small, incremental steps, but have more to do with people than servers. Like rolling Blue/Green, some people will get one version of your software while other will get another version. Unlike rolling Blue/Green, it has nothing to do with migrating servers over one at a time. Often, it is serving some small percentage of your total traffic to the new application, monitoring performance, and slowly increasing the percentage over time until all traffic is receiving the new application. If there is an error or performance issue encountered, the percentage can be immediately decreased to perform fast rollbacks. Canary deployments may also rely on a feature toggle, which turns on or off new features based on specific criteria (such as age, gender, and country of origin).

While it is certainly possible to do Rolling Blue/Green and Canary deployments with Terraform, much of the challenge depends on the kind of service you are deploying. Some managed services make this easy because the logic is baked right into the resource itself (such as the case with Azure virtual machine scale sets) while other resources need you to implement this logic yourself (such as with AWS Route 53 and AWS Application Load Balancers). In this section we will stick with the classic, and more general, Blue/Green deployment problem.

#### **9.2.1 Architecture**

Going back to the definition of Blue/Green, we will need to have two copies of the production environment, one “Blue” and the other “Green”. We will also need to have some common infrastructure that is independent of environment, such as the load balancer, VPC, and security groups. For the purposes of this exercise, I will refer to this common infrastructure as the “Base”, because it forms the foundation layer that the application will be deployed onto, similar to what we did with the two-stage deployment technique in chapters 7 and 8.

---

<sup>2</sup> Rolling Blue/Green deployments are similar to traditional Blue/Green except instead of having  $2n$  servers you only have  $n$  servers. During the deployment process you incrementally upgrade one server at a time rather than all at once.

<sup>3</sup> Canary deployments restrict releases to a subset of users. Instead of releasing to all customers everywhere, you restrict the new release to a small percentage of the target user base and increase that percentage over time.

**NOTE** Managing stateful data for Blue/Green deployments is notoriously tricky. Many people recommend including databases in the base layer, so that all production data is shared between Blue and Green.

We will be deploying version 1.0 of our software onto Green and version 2.0 onto Blue. Initially, Green will be the live server while Blue is idle. Next, we will manually cut over from Green to Blue so that Blue becomes the new live server, while Green is idle. From the user's perspective, the software update from version 1.0 to 2.0 happens instantaneous. The overarching deployment strategy is illustrated by figure 9.4

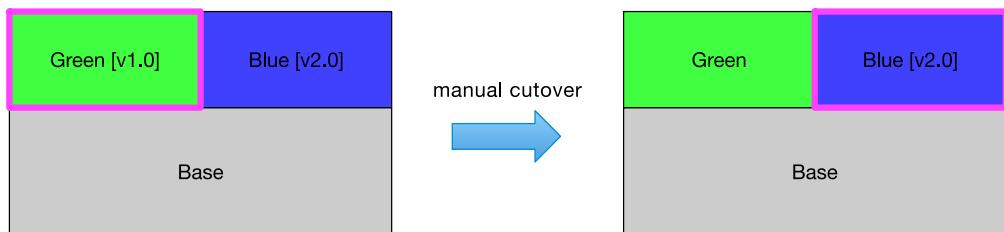


Figure 9.4 The shared, or “base”, infrastructure is deployed first. Initially, Green will be the live server, while Blue is idle. Then, a manual cutover will take place so that Blue becomes the new live server. The end result is that the customer experiences an instantaneous software update from version 1.0 to 2.0.

A detailed architecture diagram is shown in figure 9.5.

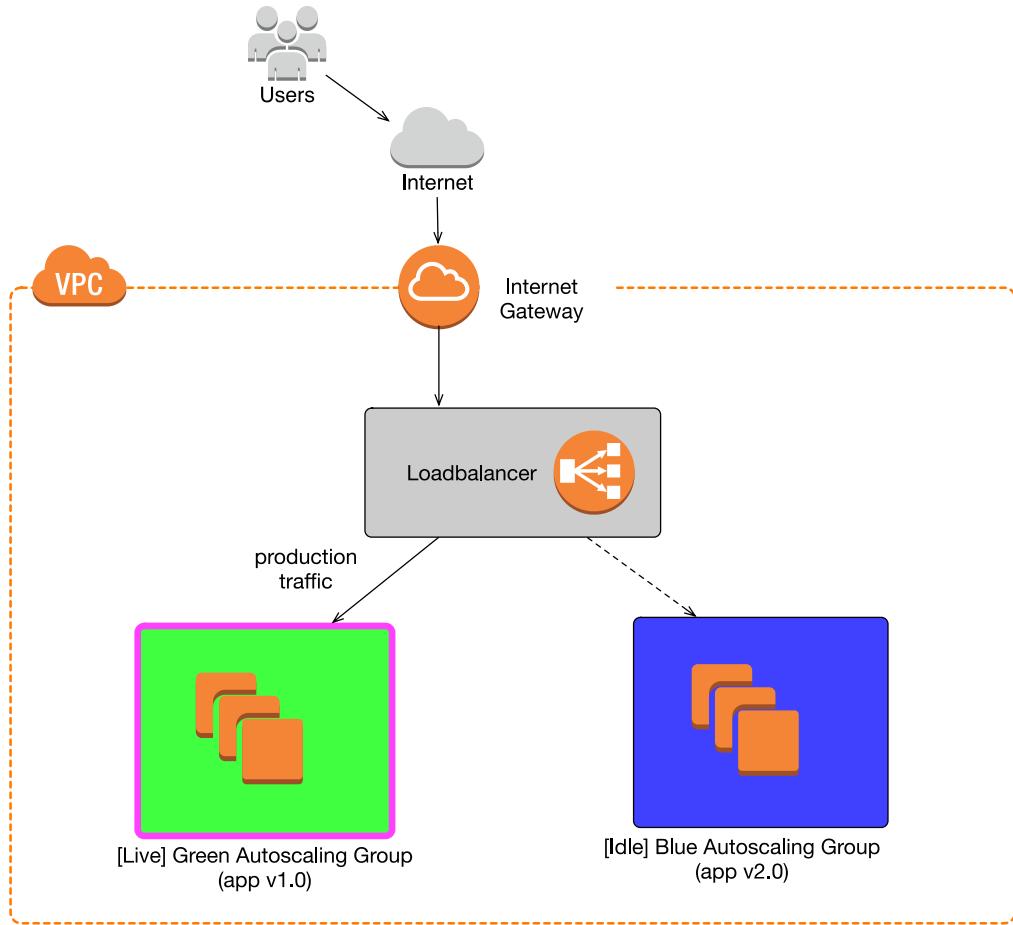


Figure 9.5 we will deploy a load balancer with two autoscaling groups. One autoscaling group will be designated as Green while this other is Blue. The load balancer serves production traffic to the current live environment.

### 9.2.2 Code

We will be leveraging pre-made modules so we can focus our attention on the big picture concepts. Create a new Terraform workspace and copy the code from Listing 9.1 into a file named `blue_green.tf`.

#### **Listing 9.1 blue\_green.tf**

```
provider "aws" {
  profile = "<profile>"
  region  = "us-west-2"
}
```

```

variable "live" {
  default = "green"
}

module "base" {
  source  = "terraform-in-action/aws/bluegreen//modules/base"
  live    = var.live
}

module "green" {
  source      = "terraform-in-action/aws/bluegreen//modules/autoscaling"
  app_version = "v1.0"
  label       = "green"
  base        = module.base
}

module "blue" {
  source      = "terraform-in-action/aws/bluegreen//modules/autoscaling"
  app_version = "v2.0"
  label       = "blue"
  base        = module.base
}

output "lb_dns_name" {
  value = module.base.lb_dns_name
}

```

**TIP** you can also use feature flags to enable/disable Blue/Green environments. For example, you could have a boolean variable called `enable_green_application` that sets the `count` attribute on a resource to either `1` or `0` (i.e. `count = var.enable_green_application ? 1 : 0`)

### 9.2.3 Deploy

Initialize the Terraform workspace with a `terraform init` and follow it up with a `terraform apply`. The result of the execution plan will be:

```
$ terraform apply
...
# module.green.module.iam_instance_profile.aws_iam_role_policy.iam_role_policy will be
  created
+ resource "aws_iam_role_policy" "iam_role_policy" {
  + id      = (known after apply)
  + name    = (known after apply)
  + policy  = jsonencode(
    {
      + Statement = [
        + {
          + Action   = "logs.*"
          + Effect   = "Allow"
          + Resource = "*"
          + Sid     = ""
        },
      ]
    + Version   = "2012-10-17"
  }
)
```

```

    + role    = (known after apply)
}

Plan: 39 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

Enter a value:

```

Confirm and wait until Terraform finishes creating the resources (approximately 5-10 minutes). The output of the apply will contain the address of the load balancer that can be used to access the current live autoscaling group. Recall that in this case, it is Green.

```

module.green.aws_autoscaling_group.webserver: Still creating... [40s elapsed]
module.green.aws_autoscaling_group.webserver: Creation complete after 42s
  [id=terraformaction-v7t08a-green-asg]
module.blue.aws_autoscaling_group.webserver: Creation complete after 48s
  [id=terraformaction-v7t08a-blue-asg]

Apply complete! Resources: 39 added, 0 changed, 0 destroyed.

Outputs:

lb_dns_name = terraformaction-v7t08a-lb-369909743.us-west-2.elb.amazonaws.com

```

Navigate to the address in the browser to pull up a simple HTML site running version 1.0 of the application on Green (see figure 9.6)



Figure 9.6 the application load balancer currently points to the green autoscaling group which hosts version 1.0 of the application

### 9.2.4 Blue/Green Cutover

Now we are ready for the manual cutover from Green to Blue. Blue is already running version 2.0 of the application, so the only thing that needs to be done is to update the load balancer listener to point from Green to Blue. In this example, it's as easy as switching the "live" variable from "green" to "blue".

#### **Listing 9.2 blue\_green.tf**

```
provider "aws" {
  profile = "<profile>"
```

```

    region  = "us-west-2"
}

variable "live" {
  default = "blue"
}

module "base" {
  source    = "terraform-in-action/aws/bluegreen//modules/base"
  live      = var.live
}

module "green" {
  source    = "terraform-in-action/aws/bluegreen//modules/autoscaling"
  app_version = "v1.0"
  label      = "green"
  base       = module.base
}

module "blue" {
  source    = "terraform-in-action/aws/bluegreen//modules/autoscaling"
  app_version = "v2.0"
  label      = "blue"
  base       = module.base
}

output "lb_dns_name" {
  value = module.base.lb_dns_name
}

```

Now run an apply again.

```
$ terraform apply
...
  ~ action {
      order          = 1
      ~ target_group_arn = "arn:aws:elasticloadbalancing:us-west-
2:215974853022:targetgroup/terraforminaction-v7t08a-blue/7e1fcf9eb425ac0a" ->
      "arn:aws:elasticloadbalancing:us-west-2:215974853022:targetgroup/terraforminaction-
v7t08a-green/80db7ad39adc3d33"
      type          = "forward"
    }

    condition {
      field  = "path-pattern"
      values = [
        "/stg/*",
      ]
    }
  }

Plan: 0 to add, 2 to change, 0 to destroy.
```

**Do you want to perform these actions?**  
Terraform will perform the actions described above.  
Only 'yes' will be accepted to approve.

**Enter a value:**

After you confirm the apply, it should only take a few seconds for Terraform to complete the action. Again, from the user's perspective, the change happens instantaneously with no discernable downtime. The load balancer address has not changed, all that has happened is that the load balancer is now serving traffic to the Blue autoscaling group rather than Green. If you refresh the page you will see that version 2.0 of the application is now in production and is served by the Blue environment. This can be visualized in figure 9.7



Figure 9.7 the load balancer now points to the blue autoscaling group which hosts version 2.0 of the application

### 9.2.5 Additional Considerations

We have demonstrated a simple example of how to do Blue/Green deployments with Terraform. Additional considerations you should take into account before implementing Blue/Green for your own deployments are:

- **Cost Savings** – the idle group does not need to be exactly the same as the active group. You can save money by scaling down the instance size, or the number of nodes, when not needed. All you have to do is scale up right before you make the cutover.
- **Reducing Blast Radius** – instead of having the load balancer and autoscaling groups all in the same Terraform workspace, it may be better to have three workspaces: one for Blue, one for Green, and one for Base. When performing the manual cutover, you mitigate risk by not having all your infrastructure in the same workspace.
- **Canary Deployments** – with AWS route53 you can perform canary deployments by having two load balancers and routing a percentage of the production traffic to each. Note that this may require executing a series of incremental Terraform deployments to achieve.

**WARNING!** please remember to take down your infrastructure with a `terraform destroy` before proceeding, to avoid incurring ongoing costs!

## 9.3 Configuration Management

When it comes to ZDD, it's important to ask the question: "Is Terraform even the right tool for the job?". In many cases, the answer is no. Let us consider the alternative of not performing ZDD with Terraform, and instead, entirely offloading the responsibilities of ZDD to configuration management.

The further you move further up an application stack; the more frequent changes occur. At the bottom is infrastructure, which is mostly static and unchanging. By comparison, applications deployed onto that infrastructure are extremely volatile. Although Terraform can deploy applications (as we have seen in previous chapters), it isn't particularly good at continuous deployment. By design, Terraform is an infrastructure provisioning tool, and is too slow and cumbersome to fit this role. Instead, a container orchestration platform or configuration management tool would be more appropriate. Since we have already examined application delivery with containers in the preceding two chapters, let us now consider configuration management.

Configuration management (CM) enables rapid software delivery onto existing servers. Some CM tools may be able to perform a degree of infrastructure provisioning, but none are particularly good at the task. Terraform is much better at infrastructure provisioning than any existing CM tool. Nevertheless, it's not a competition. You can achieve great results by combining the infrastructure provisioning capabilities of Terraform with the best parts of CM.

Superficially, it might seem that the innate mutability of CM clashes with the innate immutability of Terraform. But this isn't so. First, we know that Terraform is not as immutable as it claims to be, as in-place updates and local-exec provisioners are examples to the contrary. Second, CM is not as mutable as you might be led to believe. Yes, CM relies on mutable infrastructure, but applications being deployed onto that infrastructure can be done so immutably.

Terraform and CM tools do not have to be competitive with each other and can effectively be integrated into a common workflow. By employing the two-stage deployment technique, Terraform can provision the infrastructure, and CM can handle application delivery. This is illustrated by figure 9.8.

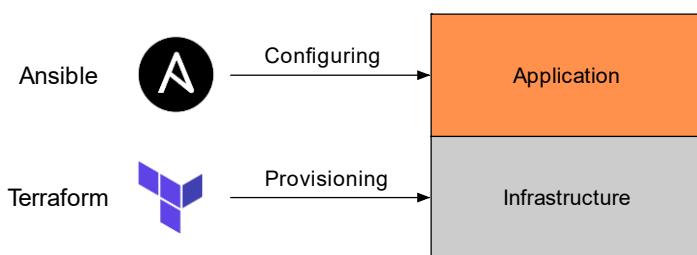


Figure 9.8 A two-stage deployment, with Terraform deploying base level infrastructure, and Ansible configuring the application.

### 9.3.1 Combining Terraform with Ansible

Ansible and Terraform make for a great pair, with HashiCorp even publicly stating that they are "better together"<sup>4</sup>. But how can these two disparate tools be successfully integrated in practice?

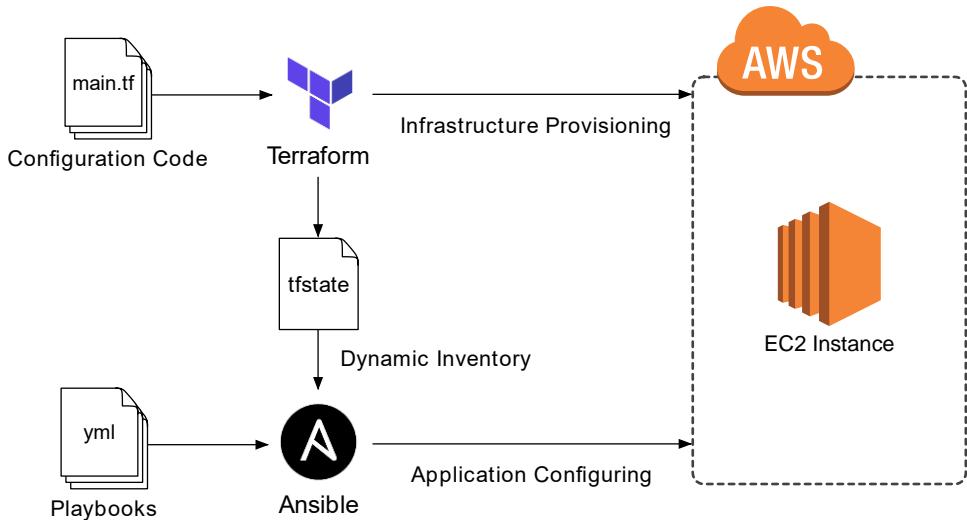
It works like this:

---

<sup>4</sup> <https://www.hashicorp.com/resources/ansible-terraform-better-together/>

1. Provision a virtual machine, or fleet of virtual machines, with Terraform.
2. Run an Ansible playbook to configure the machines and deploy new applications.

This process can be visualized by figure 9.9, when the target cloud is AWS, and the virtual machine in question is an EC2 instance.



**Figure 9.9 Terraform provisions an EC2 instance, and Ansible configures it with an Ansible playbook.**

**NOTE** Chef, Puppet, and SaltStack, could also be incorporated in a similar manner

For this scenario, we are going to provision a single EC2 instance with Terraform. The EC2 instance will have Ansible pre-installed on it and be configured with an SSH keypair generated through Terraform. Once the server is up and running, we will deploy a Nginx application onto it with Ansible. Finally, we will update the application to simulate a new application deployment.

### 9.3.2 Code

Jumping right in, we'll start by declaring the AWS provider. In a new project directory, create a `main.tf` file with the AWS provider declared at top.

#### Listing 9.3 main.tf

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

Next, we'll generate the SSH keypair that we'll use to configure the EC2 instance. The TLS provider makes this easy. After that, we'll write the private key to a local file and upload the public key to AWS (see Listing 9.4).

**NOTE** Ansible requires SSH access to push software updates. Instead of creating a new SSH keypair, you could reuse an existing one, but it's good to know how to do this with Terraform, regardless

#### Listing 9.4 main.tf

```
...
resource "tls_private_key" "key" {
  algorithm = "RSA"
}

resource "local_file" "private_key" {
  filename      = "${path.module}/ansible-key.pem"
  sensitive_content = tls_private_key.key.private_key_pem
  file_permission = "0400"
}

resource "aws_key_pair" "key_pair" {
  key_name      = "ansible-key"
  public_key    = tls_private_key.key.public_key_openssh
}
```

Configuring SSH means that we'll need to create a security group with access to port 22. Of course, we'll also need port 80 open to serve HTTP traffic. The configuration code for the AWS security group is shown in Listing 9.5.

#### Listing 9.5 main.tf

```
...
data "aws_vpc" "default" {
  default = true
}

resource "aws_security_group" "allow_ssh" {
  vpc_id = data.aws_vpc.default.id

  ingress {
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

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

  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

We'll also need to get latest Ubuntu AMI, so that we can configure the EC2 instance. This should be familiar:

#### **Listing 9.6 main.tf**

```
...
data "aws_ami" "ubuntu" {
  most_recent = true

  filter {
    name   = "name"
    values = ["ubuntu/images/hvm-ssd/ubuntu-focal-20.04-amd64-server-*"]
  }

  owners = ["099720109477"]
}
```

And now we can configure the EC2 instance itself:

#### **Listing 9.7 main.tf**

```
...
resource "aws_instance" "ansible_server" {
  ami                  = data.aws_ami.ubuntu.id
  instance_type        = "t3.micro"
  vpc_security_group_ids = [aws_security_group.allow_ssh.id]
  key_name             = aws_key_pair.key_pair.key_name

  tags = {
    Name = "Ansible Server"
  }

  provisioner "remote-exec" { #A
    inline = [
      "sudo apt update -y",
      "sudo apt install -y ansible"
    ]

    connection {
      type     = "ssh"
      user     = "ubuntu"
      host     = self.public_ip
      private_key = tls_private_key.key.private_key_pem
    }
  }

  provisioner "local-exec" { #B
    command = "ansible-playbook -u ubuntu --key-file ansible-key.pem -T 300 -i
      '${self.public_ip}', app.yml"
  }
}

#A install Ansible
#B run initial playbook
```

**NOTE** The remote-exec provisioner is exactly like a local-exec provisioner, except it first connects to a remote host

### A Case for Provisioners

I do not normally advocate the use of resource provisioners, because executing arbitrary code from Terraform is generally a bad idea, however, I feel this is one situation where an exception could be made. Instead of pre-baking an image or invoking a user-init script, a remote-exec provisioner performs simply direct, inline commands to update the system and install preliminary software. You also get the logs piped right back into Terraform stdout. Quick and easy, especially since we already have an SSH keypair on hand. But that's not the only advantage of using a remote-exec provisioner in this case. Since resource provisioners execute sequentially, we can guarantee that the local-exec provisioner running the playbook does not execute until after the remote-exec provisioner succeeds. Without a remote-exec provisioner there would be a race condition.

Lastly, we need to output a couple things, one to get the `public_ip`, and another to get the Ansible command for running the playbook.

#### Listing 9.8 main.tf

```
...
output "public_ip" {
  value = aws_instance.ansible_server.public_ip
}

output "ansible_command" {
  value = "ansible-playbook -u ubuntu --key-file ansible-key.pem -T 300 -i
          '${aws_instance.ansible_server.public_ip}', app.yml"
}
```

At this point, the Terraform is done, but we still need for a couple more files for Ansible. In particular, we will need a playbook file (`app.yml`), and an `index.html` file that will serve as our sample application.

**NOTE** if you do not already have Ansible installed on your local machine, you should do so at this point. The Ansible documentation describes how to do this<sup>5</sup>.

Create a new `app.yml` playbook file with the contents from Listing 9.9. This is a simple Ansible playbook that ensures Nginx is installed, adds an `index.html` page, and starts the Nginx service.

#### Listing 9.9 app.yml

```
---
- name: Install Nginx
  hosts: all
  become: true
```

---

<sup>5</sup> [https://docs.ansible.com/ansible/latest/installation\\_guide/intro\\_installation.html](https://docs.ansible.com/ansible/latest/installation_guide/intro_installation.html)

```

tasks:
- name: Install Nginx
  yum:
    name: nginx
    state: present

- name: Add index page
  template:
    src: index.html
    dest: /var/www/html/index.html

- name: Start Nginx
  service:
    name: nginx
    state: started

```

And here is the HTML page we'll be serving:

#### **Listing 9.10 index.html**

```

<!DOCTYPE html>
<html>
<style>
  body {
    background-color: green;
    color: white;
  }
</style>

<body>
  <h1>green-v1.0</h1>
</body>

</html>

```

Your current directory now contains the following files:

```
.
├── app.yml
├── index.html
└── main.tf
```

And, for reference, the complete contents of `main.tf` is shown in Listing 9.11.

#### **Listing 9.11 complete main.tf**

```

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

resource "tls_private_key" "key" {
  algorithm = "RSA"
}

resource "local_file" "private_key" {
  filename      = "${path.module}/ansible-key.pem"
}
```

```

sensitive_content = tls_private_key.key.private_key_pem
file_permission   = "0400"
}

resource "aws_key_pair" "key_pair" {
  key_name    = "ansible-key"
  public_key  = tls_private_key.key.public_key_openssh
}

data "aws_vpc" "default" {
  default = true
}

resource "aws_security_group" "allow_ssh" {
  vpc_id = data.aws_vpc.default.id

  ingress {
    from_port  = 22
    to_port    = 22
    protocol   = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

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

  egress {
    from_port  = 0
    to_port    = 0
    protocol   = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
}

data "aws_ami" "ubuntu" {
  most_recent = true

  filter {
    name    = "name"
    values  = ["ubuntu/images/hvm-ssd/ubuntu-focal-20.04-amd64-server-*"]
  }

  owners = ["099720109477"]
}

resource "aws_instance" "ansible_server" {
  ami                  = data.aws_ami.ubuntu.id
  instance_type        = "t3.micro"
  vpc_security_group_ids = [aws_security_group.allow_ssh.id]
  key_name             = aws_key_pair.key_pair.key_name

  tags = {
    Name = "Ansible Server"
  }
}

```

```

provisioner "remote-exec" {
  inline = [
    "sudo apt update -y",
    "sudo apt install -y ansible"
  ]

  connection {
    type      = "ssh"
    user      = "ubuntu"
    host      = self.public_ip
    private_key = tls_private_key.key.private_key_pem
  }
}

provisioner "local-exec" {
  command = "ansible-playbook -u ubuntu --key-file ansible-key.pem -T 300 -i
            '${self.public_ip}', dev.yml"
}
}

output "public_ip" {
  value = aws_instance.ansible_server.public_ip
}

output "ansible_command" {
  value = "ansible-playbook -u ubuntu --key-file ansible-key.pem -T 300 -i
            '${aws_instance.ansible_server.public_ip}', app.yml"
}

```

### 9.3.3 Infrastructure Deployment

We are now ready to deploy!

**WARNING!** Ansible must be installed on your local machine, or the local-exec provisioner will fail!

Initialize Terraform and perform a `terraform apply`:

```

$ terraform init && terraform apply -auto-approve
...
aws_instance.ansible_server: Creation complete after 2m7s [id=i-06774a7635d4581ac]

Apply complete! Resources: 5 added, 0 changed, 0 destroyed.

Outputs:

ansible_command = ansible-playbook -u ubuntu --key-file ansible-key.pem -T 300 -i
                  '54.245.143.100', app.yml
public_ip = 54.245.143.100

```

Now that the EC2 instance has been deployed, and the first Ansible playbook has run, we can view the webpage by navigating to the public IP address in the browser.

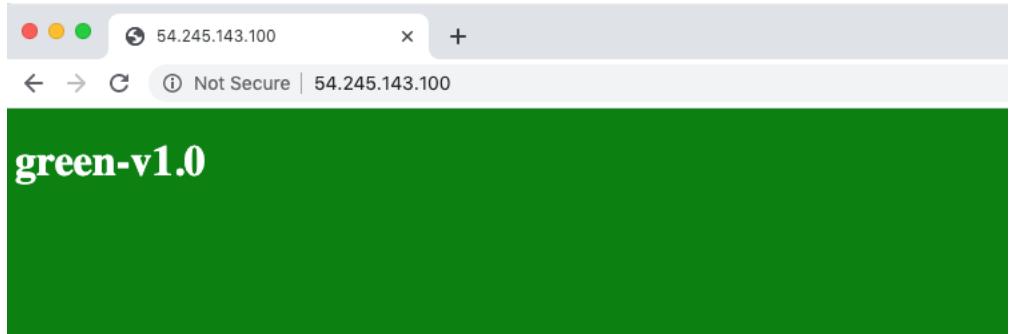


Figure 9.10 green application deployment performed with Ansible

### 9.3.4 Application Deployment

We did not need to use a local-exec provisioner to deploy the initial Ansible playbook, but it was a good example of when local-exec provisioners might be useful. Normally, application updates would be deployed independently, perhaps the result of a CI trigger. To simulate an application change, let's modify the contents of `index.html` as shown in Listing 9.12.

#### Listing 9.12

```
<!DOCTYPE html>
<html>
<style>
  body {
    background-color: blue;
    color: white;
  }
</style>

<body>
  <h1>blue-v2.0</h1>
</body>

</html>
```

By re-running the Ansible playbook, the application layer will be updated, without touching the underlying infrastructure.

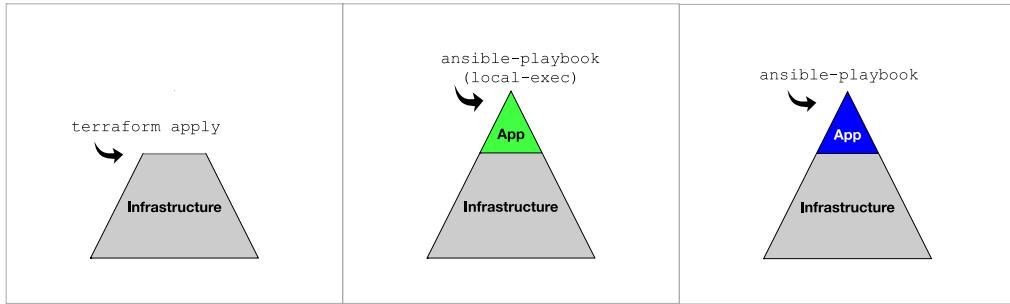


Figure 9.11 Terraform provisions initial infrastructure, while Ansible deploys applications onto that infrastructure.

Let us deploy the update now by running the Ansible playbook command from Terraform output:

**TIP** if you have more than one virtual machine, it would be better to write the addresses to a dynamic inventory file. Terraform can generate this file for you by way of string templates and the Local provider.

```
$ ansible-playbook -u ubuntu --key-file ansible-key.pem -T 300 -i '54.245.143.100,' ,  
app.yml  
  
PLAY [Install Nginx]  
*****  
*****  
*****  
  
TASK [Gathering Facts]  
*****  
*****  
*****  
ok: [54.245.143.100]  
  
TASK [Install Nginx]  
*****  
*****  
*****  
ok: [54.245.143.100]  
  
TASK [Add index page]  
*****  
*****  
*****  
changed: [54.245.143.100]  
  
TASK [Start Nginx]  
*****  
*****  
*****  
ok: [54.245.143.100]  
  
PLAY RECAP
```

```
*****
*****54.245.143.100 : ok=4    changed=1    unreachable=0    failed=0    skipped=0
rescued=0      ignored=0*****
```

Now that Ansible has redeployed our application, we can verify that changes have propagated by refreshing the webpage:

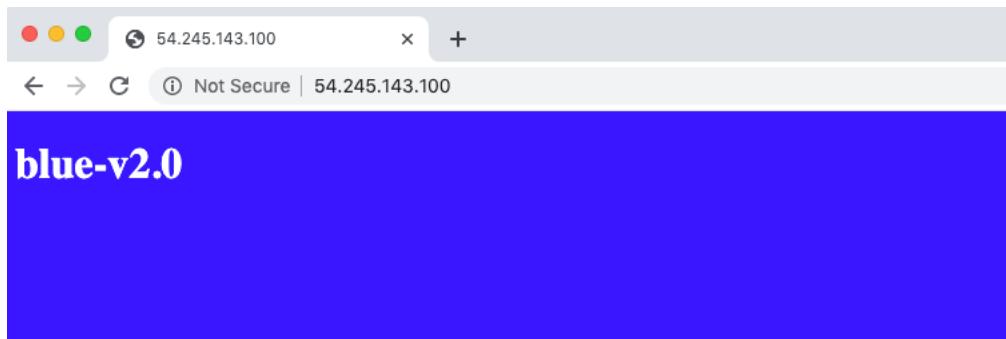


Figure 9.12 blue application deployment performed by Ansible

We have demonstrated how to combine Terraform with Ansible. Instead of worrying about how to perform ZDD with Terraform, we have offloaded the responsibility to Ansible. And we can do the same thing with any other configuration management or application delivery technology.

**WARNING!** don't forget to clean-up with a `terraform destroy`!

## 9.4 Fireside Chat

In this chapter we focused on Zero Downtime Deployments (ZDD) and what exactly that means from a Terraform perspective. We started by talking about the `lifecycle` block and how that can be used along `local-exec` health checks to ensure that a new service is running before tearing down the old one. The `lifecycle` block is the last of the resource meta-attributes; the complete list is as follows:

- `depends_on` – for specifying hidden dependencies
- `count` – for creating multiple resource instances according to a set number
- `for_each` – to create multiple instances according to a map, or set of strings
- `provider` – for selecting a non-default provider configuration
- `lifecycle` – for lifecycle customizations
- `provisioner` and `connection` – for taking extra actions after resource creation

Traditionally, ZDD refers to application deployments, either Blue/Green, rolling Blue/Green, or canary deployments. Although it's possible to use the `lifecycle` block to mimic the behavior of Blue/Green deployments, this is confusing and not recommended. Instead, we used feature flags to switch between environments with Terraform.

Lastly, we explored how to offload the responsibilities of ZDD to other, more suitable, technologies (specifically, Ansible). Yes, Terraform can deploy your entire application stack, but this isn't always convenient, nor prudent. It may be beneficial to instead use Terraform only for infrastructure provisioning, and a proven configuration management tool for application delivery. Of course, there isn't one right choice. It all depends on what you are deploying, and what serves your customers the best.

## 9.5 Summary

- The `lifecycle` block has many flags that allow for customizing resource lifecycles. Of these, the `create_before_destroy` flag is certainly the most drastic, as it completely overhauls the way Terraform fundamentally behaves.
- Performing Blue/Green deployments in Terraform is more a technique than a first-class feature. We covered one way to do Blue/Green by using feature flags to toggle between a Green and a Blue environment.
- Terraform can be combined with Ansible by using a two-stage deployment technique. In the first stage, Terraform deploys static infrastructure, while in the second stage, Ansible deploys applications on top of that infrastructure
- The TLS provider makes it easy to generate SSH keypairs. You can even write out the private key to a `.pem` file by using the Local provider.
- `remote-exec` provisioners are no different than `local-exec` provisioners, except they run on a remote machine instead of the local machine. They output to normal Terraform logs and can be used in place of `user_init` data or pre-baked AMI's.

# 10

## *Refactoring and Testing*

### This chapter covers:

- Tainting and rotating AWS access keys provisioned by Terraform
- Module expansion refactoring techniques
- Migrating state with `terraform mv` and `terraform state` commands
- Importing existing resources with `terraform import`
- Testing Infrastructure as Code with `terraform-exec`

The ancient Greek philosopher Heraclitus is famous for positing that: “Life is flux”. In other words, change is inevitable, and to resist change is to resist the essence of our existence. Perhaps nowhere is change more pronounced than in the software industry. Either due to changing customer requirements, or shifting market conditions, software is guaranteed to change. If not actively maintained, software degrades over time. Refactoring and testing are steps that developers take to keep software current.

What is refactoring? *Refactoring* is the art of improving the design of code without changing existing behavior or adding new functionality. Benefits of refactoring include:

- **Maintainability** – the ability to quickly fix bugs and address problems faced by customers.
- **Extensibility** – how easy it is to add new features. If your software is extensible, then you are more agile and able to respond to marketplace changes.
- **Reusability** – removing duplicated and highly coupled code. Reusable code is readable and easier to maintain.

Even a minor code refactoring should be thoroughly tested to ensure the system operates as intended. There are (at least) three levels of software testing to consider: unit tests, integration tests and system tests. From a Terraform perspective, we do not typically worry

about unit tests because these are already implemented at the provider level. We also don't care much about developing system tests, because they are not as well defined when it comes to Infrastructure as Code. What we do care about is *integration tests*. In other words, for a given set of inputs, does a subsystem of Terraform (i.e. a module) deploy without errors, and produce the expected output?

In this chapter we will begin by writing configuration code to self-service and rotate AWS access keys (with `terraform taint`). There are problems with the maintainability of the code, which we will improve upon in the next section by making use of *module expansions*. Module expansions are a Terraform 0.13 feature allowing the use of `count` and `for_each` on modules. They are quite powerful, and lot of old code could benefit by module expansions.

To deploy the code into production we'll need to migrate state. State migration is tedious and somewhat tricky, but as we'll see, with proper use of `terraform mv`, `terraform state`, and `terraform import`, it's achievable.

The last thing we'll investigate is how to test Terraform code with `terraform-exec`<sup>1</sup>. Terraform-exec is a HashiCorp golang library that makes it possible to programmatically execute Terraform commands. It's most similar to Gruntwork's Terratest<sup>2</sup> and allows us to write integration tests for Terraform modules.

Let's go ahead and get started.

## 10.1 Self Service Infrastructure Provisioning

Self-service is all about enabling customers to service themselves. Terraform, being a human readable configuration language, is ideal for self-service infrastructure provisioning. With Terraform, customers are able to service themselves by making pull requests (PRs) against repositories.

But wait, haven't we been doing self-service infrastructure provisioning all along? In a way, yes, but also no. This whole time we've been looking at IaC more from a developers or operations perspective, rather than a customer's perspective. Remember that not everyone has as much experience with Terraform. Creating a successful self-service model is as much designing an easy-to-use workflow, as it is the choice of technology.

Self-service infrastructure provisioning sounds great on paper, but in practice it quickly becomes chaos if rules aren't quickly established about what can and cannot be provisioned. Not only do you have to make life easy for the customer, you have to make life easy for yourself as well

<sup>1</sup> <https://github.com/hashicorp/terraform-exec>

<sup>2</sup> <https://Terratest.gruntwork.io/>



**Figure 10.1** Customers make PRs against a version-controlled source repository. This PR triggers a plan, which is reviewed by a management team. When the PR is merged, an apply runs, and the resources are deployed.

Suppose you are part of a public cloud team that is responsible for gating AWS access to teams and service accounts within the company. In this arrangement, employees are not allowed to provision AWS IAM users, policies, or access keys themselves; everything must be approved by the public cloud team. In the past, such requests may have been come through an internal IT ticketing system, but this is slower and (of course) not self-service. By storing your infrastructure as code, customers are able to directly make PRs with the changes they want. Reviewers only need to examine the result of a `terraform plan` before approving. In chapter 13, we will even see how even this minuscule governance task can be automated with Sentinel policies. For now, we'll assume this is a purely manual process.

### 10.1.1 Architecture

Let's make a self-service IAM platform. What it needs to do is provision AWS IAM users, policies and access keys with Terraform, and output a valid AWS credentials file. The module structure we'll go with is a flat module design, meaning there will be a lot of little files, an no nested modules. Each service will get its own separate file for declaring resources, and shared code will be put in auxiliary files (see figure 10.2).

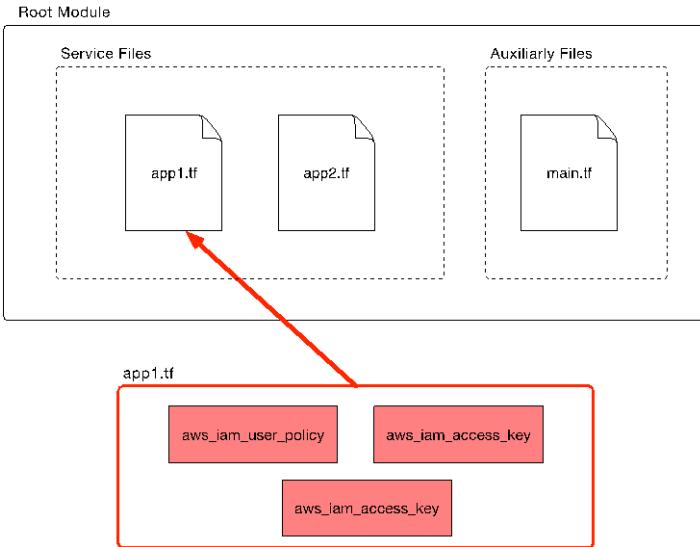


Figure 10.2 the module has two kinds of files: service and auxiliary. Service file are for keeping all managed IAM resources together for a particular service. Auxiliary files are supporting files to organize and configure the module as a whole.

### 10.1.2 Code

We'll jump right in to writing the code. Create a new directory with three files: `app1.tf`, `app2.tf` and `main.tf`. The first file, `app1.tf` contains the code for deploying an AWS IAM user called "app1-svc-account", attaches an inline policy, and provisions AWS access keys (see Listing 10.1).

#### **Listing 10.1 app1.tf**

```
resource "aws_iam_user" "app1" {
  name          = "app1-svc-account"
  force_destroy = true
}

resource "aws_iam_user_policy" "app1" {
  user      = aws_iam_user.app1.name
  policy   = <<-EOF
  {
    "Version": "2012-10-17",
    "Statement": [
      {
        "Action": [
          "ec2:Describe*"
        ],
        "Effect": "Allow",
        "Resource": "*"
      }
    ]
  }
}
```

```

        ]
    }
EOF
}

resource "aws_iam_access_key" "app1" {
    user = aws_iam_user.app1.name
}

```

The second file, `app2.tf` is similar, except it creates an IAM user called “app2-svc-account” with a policy that allows it to list S3 buckets.

### **Listing 10.2 app2.tf**

```

resource "aws_iam_user" "app2" {
    name      = "app2-svc-account"
    force_destroy = true
}

resource "aws_iam_user_policy" "app2" {
    user      = aws_iam_user.app1.name
    policy = <<-EOF
    {
        "Version": "2012-10-17",
        "Statement": [
            {
                "Action": [
                    "s3>List*"
                ],
                "Effect": "Allow",
                "Resource": "*"
            }
        ]
    }
EOF
}

resource "aws_iam_access_key" "app2" {
    user = aws_iam_user.app2.name
}

```

In `main.tf` we have a `local_file` resource that creates a valid AWS credentials file<sup>3</sup>.

### **Listing 10.3 main.tf**

```

terraform {
    required_version = "~> 0.13"
    required_providers {
        aws  = "~> 3.6"
        local = "~> 1.4"
    }
}

```

---

<sup>3</sup> <https://docs.aws.amazon.com/cli/latest/userguide/cli-configure-files.html>

```

    }

provider "aws" {
  profile = "<profile>"
  region = "us-west-2"
}

resource "local_file" "credentials" { #A
  filename      = "credentials"
  file_permission = "0644"
  sensitive_content = <<-EOF
  ${aws_iam_user.app1.name}
  aws_access_key_id = ${aws_iam_access_key.app1.id}
  aws_secret_access_key = ${aws_iam_access_key.app1.secret}

  ${aws_iam_user.app2.name}
  aws_access_key_id = ${aws_iam_access_key.app2.id}
  aws_secret_access_key = ${aws_iam_access_key.app2.secret}
  EOF
}

```

#A this will output a valid AWS credentials file

**NOTE** provider declarations are normally put in `providers.tf` and terraform settings are normally put in `versions.tf`. Here we have not done so to conserve space.

### 10.1.3 Preliminary Deployment

Deployment is easy. Initialize with `terraform init`, and deploy with `terraform apply`:

```
$ terraform apply -auto-approve
...
aws_iam_access_key.app2: Creation complete after 3s [id=AKIATESI2XGPIHJZPZFB]
local_file.credentials: Creating...
local_file.credentials: Creation complete after 0s
[id=e726f407ee85ca7fedd178003762986eae1d7a27]

Apply complete! Resources: 7 added, 0 changed, 0 destroyed.
```

After the apply completes you will have two new sets of IAM users with inline policies and access keys.

The screenshot shows the AWS IAM Management Console. In the search bar, the text 'SVC' is entered. Below the search bar, there are buttons for 'Add user' and 'Delete user'. On the left sidebar, under 'Access management', the 'Users' option is selected. The main table displays two results:

User name	Groups	Access key age	Password age	Last activity	MFA
app1-svc-account	None	Today	None	None	Not enabled
app2-svc-account	None	Today	None	None	Not enabled

**Figure 10.3** Terraform has provisioned two new IAM users with inline policies and created access keys for those users.

An AWS credentials file has also been generated using `local_file`. The credentials file can be used to authenticate to the AWS CLI.

```
$ cat credentials
[app1-svc-account]
aws_access_key_id = AKIATESI2XGPIUSUHWUV
aws_secret_access_key = 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

[app2-svc-account]
aws_access_key_id = AKIATESI2XGPIHJZPZFB
aws_secret_access_key = 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
```

**NOTE** instead of writing secrets in plain text to a credentials file, it's better to store these values in a centralized secrets management tool, like HashiCorp Vault or AWS Secrets Manager. We will cover this more in chapter 13.

There are only two service accounts being managed by Terraform at the moment, but it's easy to imagine how more service accounts could be provisioned. All that needs be done is create a new service file and update `local_file`. Although the code works, there are some problems that emerge when scaling up. Before we discuss what improvements could be made, let's first rotate access keys with `terraform taint`.

#### 10.1.4 Tainting and Rotating Access Keys

Regular secrets rotation is a well-known security best practice. Even the ancient Romans knew this; sentries would change camp passwords once a day. Since access keys allow service accounts to provision resources in AWS accounts, it's a good idea to rotate these as frequently as possible (at least once every 90 days).

Although we could rotate access keys by performing a `terraform destroy` followed by `terraform apply`, sometimes you wouldn't want to do this. For example, if there was a

permanent resource fixture, such as an RDS database or S3 bucket included as part of the deployment, a `terraform destroy` would delete these and result in data loss.

We can target the destruction and recreation of individual resources with the `terraform taint` command. During the next apply the resource will be destroyed and created anew.

The usage of the command is as follows:

```
terraform taint [options] address
```

**NOTE** address is the `resource address`<sup>4</sup> that uniquely identifies a resource within a given configuration

To rotate access keys, first list the resources in the state file to obtain resource addresses. The command `terraform state list` does this for us.

```
$ terraform state list
aws_iam_access_key.app1
aws_iam_access_key.app2
aws_iam_user.app1
aws_iam_user.app2
aws_iam_user_policy.app1
aws_iam_user_policy.app2
local_file.credentials
```

It looks like our two resource addresses are: `aws_iam_access_key.app1` and `aws_iam_access_key.app2`. Go ahead and taint these resources so they can be recreated during the next apply:

```
$ terraform taint aws_iam_access_key.app1
Resource instance aws_iam_access_key.app1 has been marked as tainted.
$ terraform taint aws_iam_access_key.app2
Resource instance aws_iam_access_key.app2 has been marked as tainted.
```

When we run a `terraform plan`, we can see that the `aws_access_key` resources have been marked as tainted, and will be recreated:

```
$ terraform plan
...
Terraform will perform the following actions:

# aws_iam_access_key.app1 is tainted, so must be replaced
-/+ resource "aws_iam_access_key" "app1" {
    + encrypted_secret = (known after apply)
    ~ id               = "AKIAATESI2XGPIUSUHWUV" -> (known after apply)
    + key_fingerprint = (known after apply)
    ~ secret           = "1qETH8vetvdV8gv00+d1A0jvuXh7qHiQRh0tEmaY" -> (known after
        apply)
    ~ ses_smtp_password = "AiLTGCR7lNIM1u8Pl3cTOHu10Ni5JbhxULGdb+4z6inL" -> (known after
        apply)
```

---

<sup>4</sup> <https://terraform.io/docs/internals/resource-addressing.html>

```

    ~ status          = "Active" -> (known after apply)
    user           = "app1-svc-account"
}

...
Plan: 3 to add, 0 to change, 3 to destroy.

```

**NOTE** if you ever taint the wrong resource, you can always undo your mistake with the complementary command:`terraform untaint`

If we apply changes, the access keys will be recreated without affecting anything else, (except, of course, dependent resources like `local_file`). Apply changes now by running `terraform apply`.

```

$ terraform apply -auto-approve
...
aws_iam_access_key.app1: Creation complete after 0s [id=AKIATESI2XGPIQGHRH5W]
local_file.credentials: Creating...
local_file.credentials: Creation complete after 1s
[id=ea6994e2b186bbd467cceee89ff39c10db5c1f5e]

Apply complete! Resources: 3 added, 0 changed, 3 destroyed.

```

We can verify that the access keys have indeed been rotated by cat-ing the credentials file and observing that it has new access and secret access keys.

```

$ cat credentials
[app1-svc-account]
aws_access_key_id = AKIATESI2XGPIQGHRH5W
aws_secret_access_key = 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

[app2-svc-account]
aws_access_key_id = AKIATESI2XGPLJNKW5FC
aws_secret_access_key = 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

```

## 10.2 Refactoring Terraform Configuration

While the code may be suitable for the current use case, there are deficiencies that will result in long-term maintainability issues, namely:

- **Duplicated Code** – As new users and policies are provisioned, correspondingly more service files are required. This means a lot of copy/paste.
- **Name Collisions** – Because of all the copy/paste, name collisions on resources are practically inevitable. You'll waste time by resolving silly name conflicts.
- **Inconsistency** – As the code base grows, it becomes harder and harder to maintain uniformity, especially if pull requests are being made by people who aren't Terraform experts

To alleviate the above concerns, we need to refactor.

### 10.2.1 Modularizing Code

The biggest refactoring improvement we can make it to put reusable code into modules. Not only does this solve the problem of duplicated code (i.e. resources in modules only have to be declared once), it also solves the problem of name collisions (resources do not conflict with resources in other modules) and inconsistency (hard to mess up a PR if there's not much code that's being changed).

The first step to modularizing an existing workspace is to identify opportunities for code reuse. Comparing `app1.tf` with `app2.tf`, you'll notice the same three resources being declared in both: an IAM user, IAM policy, and IAM access key (see Snippets 10.1 and 10.2).

#### **Snippet 10.1 app1.tf**

```
resource "aws iam user" "app1" {
  name      = "app1-svc-account"
  force_destroy = true
}

resource "aws_iam_user_policy" "app1" {
  user    = aws_iam_user.app1.name
  policy = <<-EOF
  {
    "Version": "2012-10-17",
    "Statement": [
      {
        "Action": [
          "ec2:Describe*"
        ],
        "Effect": "Allow",
        "Resource": "*"
      }
    ]
  }
  EOF
}

resource "aws_iam_access_key" "app1" {
  user = aws_iam_user.app1.name
}
```

#### **Snippet 10.2 app2.tf**

```
resource "aws iam user" "app2" {
  name      = "app2-svc-account"
  force_destroy = true
}

resource "aws_iam_user_policy" "app2" {
  user    = aws_iam_user.app1.name
  policy = <<-EOF
  {
    "Version": "2012-10-17",
    "Statement": [
      {
        "Action": [
```

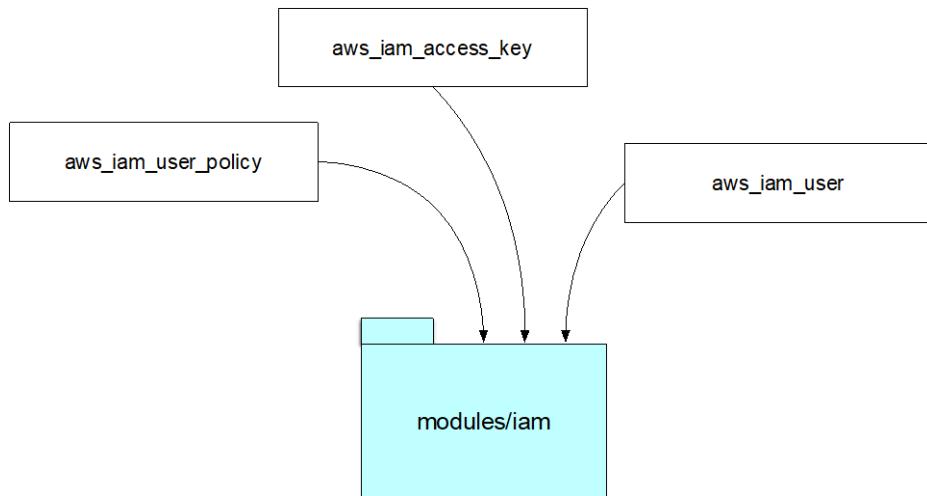
```

        "s3>List*"
    ],
    "Effect": "Allow",
    "Resource": "*"
}
]
EOF
}

resource "aws iam access key" "app2" {
  user = aws_iam_user.app2.name
}

```

There are slight differences between the policy configurations, of course, but this can be easily parameterized. We'll move the three resources into a common module called "iam" (see figure 10.4).



**Figure 10.4 Consolidating common IAM resources in a Terraform module**

Next, we'll need to clean up `main.tf`. This file is responsible for provisioning a credentials text document containing the AWS access and secret access keys, but the way it does so is inefficient, as it requires explicitly referencing each resource.

#### Snippet 10.3 local\_file declaration

```

resource "local_file" "credentials" {
  filename      = "credentials"
  file_permission = "0644"
  sensitive_content = <<-EOF
  ${aws_iam_user.app1.name}] #A
  aws_access_key_id = ${aws_iam_access_key.app1.id} #A

```

```

aws_secret_access_key = ${aws_iam_access_key.app1.secret} #A

[$aws_iam_user.app2.name] #B
aws_access_key_id = ${aws_iam_access_key.app2.id} #B
aws_secret_access_key = ${aws_iam_access_key.app2.secret} #B
EOF
}

```

#A explicitly references app1 resources

#B explicitly references app2 resources

Each time a new IAM user is provisioned, you'll need to update this file. At first this may not seem like a big deal, but over time it quickly becomes a hassle. Here would be a good place to use template strings. A three-line snippet with profile name, AWS access key id, and AWS secret access key can be produced by the IAM module and dynamically joined with other such snippets to form a credentials document.

### 10.2.2 Module Expansions

Consider what the interface for the IAM module should be. At the very least, it should accept two input parameters, one to assign a service name, and another to attach a list of policies. Accepting a list of policies is better than how we previously had it – only being able to attach a single policy. Our module will also produce an output with a 3-line template string that we can join together with other strings (see figure 10.5).

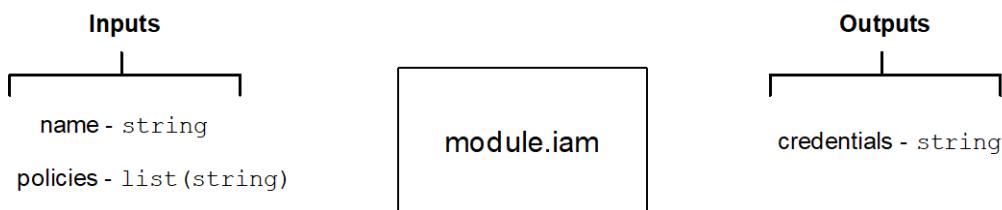


Figure 10.5 Inputs and outputs for IAM module

Until recently, we would have to declare each instance of a module like this separately, as shown in Snippet 10.4.

#### Snippet 10.4 module declarations

```

module "iam-app1" { #A
  source  = "./modules/iam"
  name    = "app1"
  policies = [file("./policies/app1.json")]
}

module "iam-app2" { #A
  source  = "./modules/iam"
  name    = "app2"
}

```

```
policies = [file("./policies/app2.json")]
}
```

#A two instances of the same module used to have to be declared separately

This isn't terrible, but it also isn't ideal. Even though we've modularized the code, there's still copy/paste occurring each time you want a new module instance. It diminished a lot of the benefit of using nested modules, and was a major reason why many people favored using flat modules.

Fortunately, there is now a solution. With the advent of Terraform 0.13, a new feature was released called *module expansions*. Module expansions make it possible to use `count` and `for_each` on a module in the same way you could for a resource. Instead of declaring modules multiple times, now you only have to declare it once. For example, assuming we had a map of configuration stored in a `local.policy_mapping` value, Figure 10.6 shows how a single module declaration could expand into multiple instances.

### Source Code

```
module "iam" {
  source  = "./modules/iam"
  for_each = local.policy_mapping
  name    = each.key
  policies = each.value.policies
}
```

Expands to

### Module Expansion

```
module.iam["app1"] {
  source  = "./modules/iam"
  name    = "app1"
  policies = [local.policies["app1.json"]]
}

module.iam["app2"] {
  source  = "./modules/iam"
  name    = "app2"
  policies = [local.policies["app2.json"]]
}
```

Figure 10.6 Expanding a Terraform module with `for_each`

Like `for_each` on resources, `for_each` on a module requires providing configuration via either a set or a map. Here we will use a map, with the keys being the `name` attribute, and the values being an object with a single attribute called `policies`. Policies will be of type `list(string)` and contains the JSON policy documents for each policy that will be attached to the IAM user. The code is shown in Listing 10.4.

#### **Listing 10.4 main.tf**

```
locals {
  policy_mapping = {
    "app1" = {
      policies = [local.policies["app1.json"]],
    },
    "app2" = {
      policies = [local.policies["app2.json"]],
    }
}
```

```

    policies = [local.policies["app2.json"]],
},
}
}

module "iam" {
  source  = "./modules/iam"
  for_each = local.policy_mapping #A
  name     = each.key
  policies = each.value.policies
}

```

#A module expansion creates a separate instance for each element of for\_each

### Why Not Use Sets?

I recommend using maps instead of sets whenever you have more than one attribute that needs to be set on a module. Maps allow you to pass entire objects, whereas sets do not. Moreover, you can only pass in a set of type `set(string)`, meaning you would have to awkwardly encode data in the form of a JSON string and then decode it with `jsondecode()` if you wanted to pass more than a single attribute worth of data. Not only is this cumbersome, it also makes the plan messier by spitting out a lot of unnecessary information and making resource addresses (strings that reference a specific resource) longer than they should be. Of course, you could choose to use `count` with set, but count indices have their own problems. Overall, I cannot recommend the use of sets with module expansions, unless you have only a single attribute that needs to be set.

### 10.2.3 Replacing Multi-Line Strings with Local Values

We are refactoring an existing Terraform workspace to aid in readability and maintainability. One of the key aspects that goes into this is to how to make it easy for someone to configure the workspace inputs. Remember that we have two module inputs: `name` (pretty self-explanatory) and `policies` (which needs some further explanation). In this case, `policies` is an input variable of type `list(string)`, designed to accept a list JSON policy documents to attach to an individual IAM user. We have a choice in how we can do this; we can either embed the policy documents inline as string literals (which is what we've been doing) or read the policy documents from an external file (the better option of the two).

The reason why embedding string literals, especially multi-line string literals, is generally a bad idea, is because it hurts readability. Having too many string literals in Terraform configuration makes it messy and hard to find what you're looking for. Better just to keep this information in a separate file, and read from it using either `file()` or `fileset()`. In Listing 10.5, I've made leveraged a for-expression to produce a map of key-value pairs containing the filename and contents of each policy file. That way, policies can be stored together in a common directory and be fetched by filename.

#### **Listing 10.5 main.tf**

```
locals {
  policies =
```

```

    for path in fileset(path.module, "policies/*.json") : basename(path) =>
        file(path)
    }
policy_mapping = {
    "app1" = {
        policies = [local.policies["app1.json"]],
    },
    "app2" = {
        policies = [local.policies["app2.json"]],
    },
}
}

module "iam" {
    source  = "./modules/iam"
    for_each = local.policy_mapping
    name     = each.key
    policies = each.value.policies
}
}

```

To give you an idea what the fancy for-expression does, the calculated value of `local.policies`, which is the result of the for-expression, is shown in Snippet 10.5.

#### Snippet 10.5 `local.policies` value

```
{
    "app1.json" = "{\n        \"Version\": \"2012-10-17\",\\n        \"Statement\": [\n            {\n                \"Action\": [\n                    \"ec2:Describe*\"\n                ],\\n                \"Effect\":\n                \"Allow\",\\n                \"Resource\": \"*\"\n            }\\n        ]\\n    }\n    \"app2.json\" = "{\n        \"Version\": \"2012-10-17\",\\n        \"Statement\": [\n            {\n                \"Action\": [\n                    \"s3>List*\"\n                ],\\n                \"Effect\":\n                \"Allow\",\\n                \"Resource\": \"*\"\n            }\\n        ]\\n    }\n"
}
```

As you can see, we can now reference the JSON policy documents for individual policies by filename. For example, `local.policies["app1.json"]` would return the contents of `app1.json`. Now all we need to do is make sure these files actually exist.

Create a `policies` folder in the current working directory. In this folder, create two new files, `app1.json` and `app2.json`. The contents of the files are shown in Listings 10.6 and 10.7, respectively.

#### Listing 10.6 `app1.json`

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
                "ec2:Describe*"
            ],
            "Effect": "Allow",
            "Resource": "*"
        }
    ]
}
```

**Listing 10.7 app2.json**

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "s3>List*"
      ],
      "Effect": "Allow",
      "Resource": "*"
    }
  ]
}
```

**10.2.4 Looping Through Multiple Module Instances**

Remember how the IAM module returns a `credentials` output? This will be a little three-line string that can be appended together with other such strings to produce a complete and valid AWS credentials file. The `credentials` string will be of the form shown in Snippet 10.6.

**Snippet 10.6 Sample module.iam[app1.json].output**

```
[app1-svc-account]
aws_access_key_id = AKIATESI2XGPIQGHRH5W
aws_secret_access_key = 8x4NAEPOfmvfa9YIeL0QgPFT4iyTIisfv+svMNrn
```

If each module instance produces its own output, we can join them together with the build-in `join()` function. The for-expression shown in Snippet 10.7 loops through each instance of the `module.iam` expansion, accesses the `credentials` output, and joins them together with a newline.

**Snippet 10.7 join() for-expression**

```
join("\n", [for m in module.iam : m.credentials])
```

**Splat Expressions Operate Only on Lists**

A splat expression is syntactic sugar allowing you to concisely express simple for-expressions. For example, if you had a list of objects each with the `id` attribute, you could extract all `ids` in a new list of strings with the following expression: `[for v in var.list : v.id]`. In contrast, the splat expression: `var.list[*].id` is far more concise (note: the special `*` symbol indicates iterating over all elements of a list).

Although convenient, splat expressions are less useful than they could be since they only operate on lists. If they were able to operate on maps, you could use them to reference resources or modules created with `for_each`. For instance, the preceding for-expression `[for m in module.iam : m.credentials]` could be replaced with `module.iam[*].credentials`. Other than for historical reasons, I am not sure why this isn't already possible. It's disappointing that splat expressions don't work the same for maps as they do for lists.

The complete `main.tf` code, with included Terraform settings block, and provider declarations, is shown in Listing 10.8.

#### **Listing 10.8 main.tf**

```
terraform {
  required_version = "~> 0.13"
  required_providers {
    aws   = "~> 3.6"
    local = "~> 1.4"
  }
}

provider "aws" {
  profile = "<profile>"
  region = "us-west-2"
}

locals {
  policies = {
    for path in fileset(path.module, "policies/*.json") : basename(path) => file(path)
  }
  policy_mapping = {
    "app1" = {
      policies = [local.policies["app1.json"]],
    },
    "app2" = {
      policies = [local.policies["app2.json"]],
    },
  }
}

module "iam" { #A
  source  = "./modules/iam"
  for_each = local.policy_mapping
  name    = each.key
  policies = each.value.policies
}

resource "local_file" "credentials" {
  filename = "credentials"
  content  = join("\n", [for m in module.iam : m.credentials])
}
```

#A the iam module doesn't exist yet, but it will soon

#### **10.2.5 New IAM Module**

Now it's time to implement the IAM module that will deploy three IAM resources (user, policy, and access key). This module will have input two variables (`name` and `policy`) and one output value (`credentials`). Create a file with relative path `./modules/iam/main.tf` and insert the following code from Listing 10.9.

**NOTE** a standard module structure would have code split into `main.tf`, `variables.tf` and `outputs.tf`; again, I've not done this for the sake of brevity

**Listing 10.9 main.tf**

```

variable "name" {
  type = string
}

variable "policies" {
  type = list(string)
}

resource "aws_iam_user" "user" {
  name          = "${var.name}-svc-account"
  force_destroy = true
}

resource "aws_iam_policy" "policy" { #A
  count   = length(var.policies)
  name    = "${var.name}-policy-${count.index}"
  policy   = var.policies[count.index]
}

resource "aws_iam_user_policy_attachment" "attachment" {
  count   = length(var.policies)
  user    = aws_iam_user.user.name
  policy_arn = aws_iam_policy.policy[count.index].arn
}

resource "aws_iam_access_key" "access_key" {
  user = aws_iam_user.user.name
}

output "credentials" { #B
  value = <<-EOF
  [${aws_iam_user.user.name}]
  aws_access_key_id = ${aws_iam_access_key.access_key.id}
  aws_secret_access_key = ${aws_iam_access_key.access_key.secret}
  EOF
}

#A support for attaching multiple policies
#B three-line template string

```

At this point we are now code complete. Your completed project should contain the following files:

```

.
├── credentials
├── main.tf
└── modules
    └── iam
        └── main.tf
└── policies
    ├── app1.json
    └── app2.json
└── terraform.tfstate

3 directories, 6 files

```

## 10.3 Migrating Terraform State

After re-initializing the workspace with a `terraform init`, a `terraform plan` reveals that Terraform intends to destroy and recreate all resources during the subsequent `terraform apply`:

```
$ terraform plan
...
# module.iam["app2"].aws_iam_user.user will be created
+ resource "aws_iam_user" "user" {
    + arn          = (known after apply)
    + force_destroy = true
    + id           = (known after apply)
    + name         = "app2-svc-account"
    + path          = "/"
    + unique_id     = (known after apply)
}

# module.iam["app2"].aws_iam_user_policy_attachment.attachment[0] will be created
+ resource "aws_iam_user_policy_attachment" "attachment" {
    + id          = (known after apply)
    + policy_arn  = (known after apply)
    + user        = "app2-svc-account"
}
Plan: 9 to add, 0 to change, 7 to destroy. #A
-----
```

#A All resources will be destroyed and recreated

The reason this happens is because Terraform does not know that resources declared in the IAM module are the same as previously provisioned resources. Oftentimes, it isn't an issue having resources be destroyed and recreated; however, it is if it results in significant data loss. For example, if you had a deployed database, you would certainly want to avoid deleting it. For the IAM scenario, we do not have any database, but let's say that we want to avoid deleting IAM users, because perhaps the associated AWS CloudWatch logs are important. We'll skip migrating IAM policies or access keys, because there is nothing special about these.

Unfortunately for us, Terraform state migration is rather difficult and tedious. It's difficult because it requires intimate knowledge about how state is stored and it's tedious because – although not entirely manual – it would take a long time to migrate more than a handful of resources.

**NOTE** HashiCorp has announced that improved imports could be a deliverable of Terraform 1.0<sup>5</sup>. Hopefully this will alleviate the worst sufferings of state migration.

### 10.3.1 State File Structure

Let us now consider what goes into Terraform state. If you recall from chapter 2, state contains information about what is currently deployed and is automatically generated from configuration code as part of `terraform apply`. To migrate state, we need to move or import resources into a correct destination resource address (see figure 10.7.)

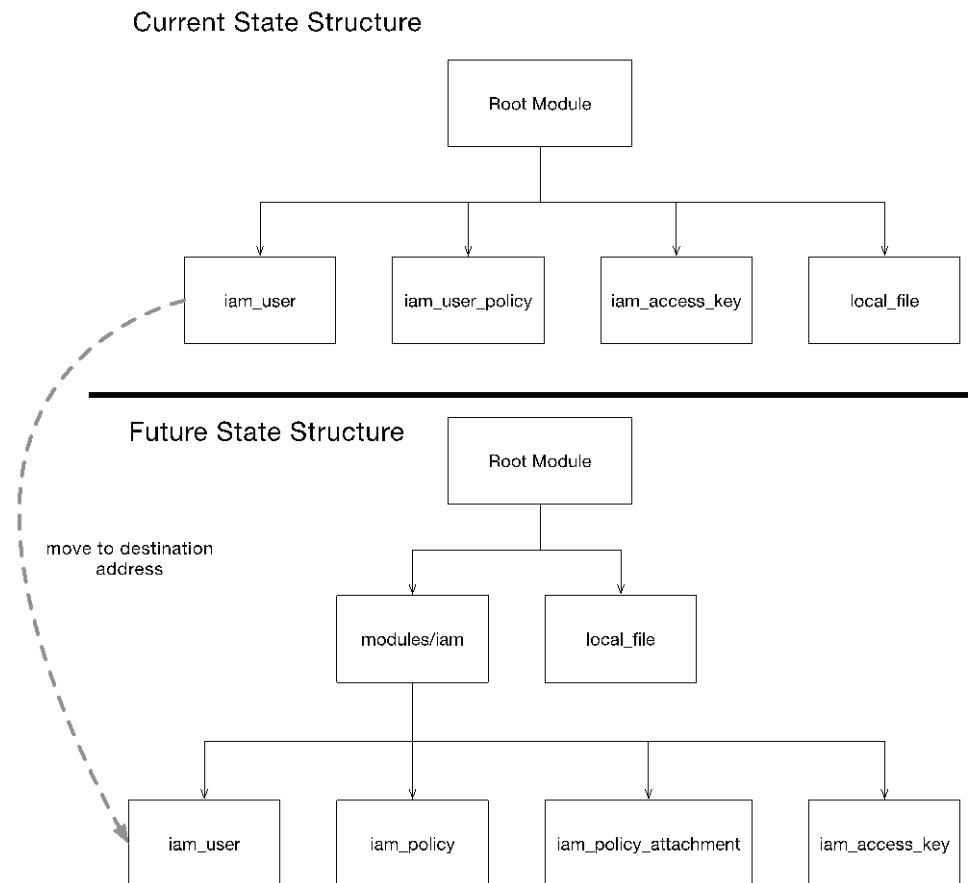


Figure 10.7 Current vs. future structure of state file. We want to move resources from where they were in the old

<sup>5</sup> <https://www.hashicorp.com/resources/the-path-to-terraform-1-0>

configuration to where they will be in the new version. This will prevent the resource from being destroyed and recreated during the next apply.

There are three options when it comes to migrating state:

1. Manually editing the state file (not recommended)
2. Moving stateful data with `terraform state mv`
3. Deleting old resources with `terraform state rm` and reimporting with `terraform import`

Of the three methods, the first is the most flexible, but also the most dangerous because of the potential for human error. Methods two and three are easier and safer. In the following two sections we will see these methods in practice.

**WARNING!** Manually editing the state file is not recommended except in niche situations, such as correcting provider errors.

### 10.3.2 Moving Resources

We have to move the existing IAM users state from their current resource address to their final resource address. We have to move them so they won't be deleted and recreated during the next apply. To accomplish this, we will use `terraform state mv` to move the resource state around. The command to move a resource (or module) into a desired destination address is:

```
terraform state mv [options] SOURCE DESTINATION
```

Source and destination here both refer to resource addresses. The source address is where the resource is currently located, and the destination address is where it will go. But how do we know what the current resource addresses is? The easiest way to find out is with `terraform state list`:

**NOTE** if you haven't already done so, perform a `terraform init` to download providers and install modules

```
$ terraform state list
aws_iam_access_key.app1
aws_iam_access_key.app2
aws_iam_user.app1
aws_iam_user.app2
aws_iam_user_policy.app1
aws_iam_user_policy.app2
local_file.credentials
```

All we need to do is move the IAM users for app1 and app2 into the "iam" module. The source address for app1 is: `aws_iam_user.app1`, the destination address for app1 is:

`module.iam["app1"]`. Therefore, to move the resource state we just need to run the following command:

```
$ terraform state mv aws_iam_user.app1 module.iam["app1"].aws_iam_user.user
Move "aws_iam_user.app1" to "module.iam["app1"].aws_iam_user.user"
Successfully moved 1 object(s).
```

Similarly, for app2:

```
$ terraform state mv aws_iam_user.app2 module.iam["app2"].aws_iam_user.user
Move "aws_iam_user.app2" to "module.iam["app2"].aws_iam_user.user"
Successfully moved 1 object(s).
```

By listing the resources in the state file again, you can verify that the resources have indeed been moved successfully:

```
$ terraform state list
aws_iam_access_key.app1
aws_iam_access_key.app2
aws_iam_user_policy.app1
aws_iam_user_policy.app2
local_file.credentials
module.iam["app1"].aws_iam_user.user
module.iam["app2"].aws_iam_user.user
```

**NOTE** you can move a resource or module to any address, even one that does not exist within your current configuration. This can cause unexpected behavior, which is why you have to be careful to get the right address.

### 10.3.3 Redeploy

Our mission was to migrate existing IAM users to their future position in Terraform state so that they won't be deleted when the configuration code is updated, based on our refactoring. We did make a stipulation that said we don't want IAM users to be deleted and recreated (because reasons) but we didn't make this condition for IAM access keys or policies, as there's really nothing important there.

A quick `terraform plan` verifies that we have indeed accomplished our mission because now only 7 resources are slated to be created, and 5 destroyed, as opposed to the 9 and 7 from earlier. This means that the two IAM users will not be destroyed and recreated, as they are already in their correct position.

```
$ terraform plan
...
# module.iam["app2"].aws_iam_user_policy_attachment.attachment[0] will be created
+ resource "aws_iam_user_policy_attachment" "attachment" {
  + id      = (known after apply)
  + policy_arn = (known after apply)
  + user    = "app2-svc-account"
}

Plan: 7 to add, 0 to change, 5 to destroy.
```

We can now apply the changes with confidence, knowing that our state migration has been accomplished:

```
$ terraform apply -auto-approve
...
module.iam["app2"].aws_iam_user_policy_attachment.attachment[0]: Creation complete after 2s
  [id=app2-svc-account-20200929075715719500000002] local_file.credentials: Creating...
local_file.credentials: Creation complete after 0s
  [id=270e9e9b124fdf55e223ac263571e8795c5b6f19]

Apply complete! Resources: 7 added, 0 changed, 5 destroyed.
```

### 10.3.4 Importing Resources

The other way Terraform state can be migrated is by deleting and reimporting resource. Resources can be deleted with `terraform state rm` and imported with `terraform import`. Deleting resources is fairly self-explanatory, it will be removed from the state file, but importing resources requires some further explanation. Resource imports are how unmanaged resources are converted into managed resources. For example, if you created resources out-of-band, such as through the CLI, or another IaC tool like CloudFormation, you could import them into Terraform as managed resources. `terraform import` is to unmanaged resources what `terraform refresh` is to managed resources. We will use `terraform import` to reimport a deleted resource into the correct resource address (not a traditional use case, I'll grant, but a useful teaching exercise nonetheless).

**NOTE** check with the relevant Terraform provider documentation to ensure imports are allowed for a given resource

Let's first remove the IAM user from Terraform state so we can reimport it. The usage of the `remove` command is as follows:

```
terraform state rm [options] ADDRESS
```

This command allows you to remove specific resources/modules from Terraform state. I normally use it for fixing corrupted state, such as when buggy resources are preventing you from applying or destroying the rest of your configuration code.

**TIP** Corrupted state is usually the result of buggy provider source code, and you should file a support ticket on the corresponding GitHub repository if you ever have this happen to you.

Before we remove the resource from state, we need the ID so we can reimport it later.

```
$ terraform state show module.iam["app1"].aws_iam_user.user
# module.iam["app1"].aws_iam_user.user:
resource "aws_iam_user" "user" {
  arn          = "arn:aws:iam::215974853022:user/app1-svc-account"
  force_destroy = true
  id           = "app1-svc-account"
  name         = "app1-svc-account"
```

```

path      = "/"
tags     = {}
unique_id = "AIDATESI2XGPBXYYGHJ0O"
}

```

The ID value for this resource is the name of the IAM user, in this case “app1-service-account”. A resource’s ID is set at the provider level, and is not always what you think it should be, but it is guaranteed to be unique. You can see what it is with `terraform show`, or figure it out yourself by reading through provider documentation.

Let’s delete the app1 IAM user from state by deleting it with the `terraform state rm` command, and passing in the resource ID from `terraform state show`:

```

$ terraform state rm module.iam["app1"].aws_iam_user.user
Removed module.iam["app1"].aws_iam_user.user
Successfully removed 1 resource instance(s).

```

Now Terraform is not managing the IAM resource and doesn’t even know that it exists. If we were to run another `apply`, Terraform would attempt to create an IAM user with the same name, which would actually cause a name conflict error, as you cannot have two IAM users with the same name in AWS. What we need to do is import the resource into the desired location, and bring it back under the yolk of Terraform. We can do that with `terraform import`.

The usage of the command is:

```
terraform import [options] ADDRESS ID
```

`ADDRESS` is the destination resource address of where you want your resource to be imported to (requires configuration to be present for this to work), and `ID` is the unique resource ID (“app1-service-account”). Import the resource now with `terraform import`:

```

$ terraform import module.iam["app1"].aws_iam_user.user app1-svc-account
module.iam["app1"].aws_iam_user.user: Importing from ID "app1-svc-account"...
module.iam["app1"].aws_iam_user.user: Import prepared!
  Prepared aws_iam_user for import
module.iam["app1"].aws_iam_user.user: Refreshing state... [id=app1-svc-account]

```

```
Import successful!
```

```
The resources that were imported are shown above. These resources are now in
your Terraform state and will henceforth be managed by Terraform.
```

An interesting thing to point out is that the state we import doesn’t match what our configuration is. In fact, if you were to do a `terraform plan` it will suggest performing an update-in-place:

```

$ terraform plan
...
An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  ~ update in-place

```

```
Terraform will perform the following actions:
```

```
# module.iam["app1"].aws_iam_user.user will be updated in-place
~ resource "aws_iam_user" "user" {
    arn          = "arn:aws:iam::215974853022:user/app1-svc-account"
    + force_destroy = true
    id           = "app1-svc-account"
    name         = "app1-svc-account"
    path         = "/"
    tags          = {}
    unique_id    = "AIDATESI2XGPBXYYGHJ00"
}
```

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

If you inspect the state file, you'll notice that `force_destroy` attribute is set to `null` instead of `true` (which is what it should be).

### Snippet 10.8 excerpt from terraform.tfstate

```
...
  {
    "module": "module.iam[\"app1\"]",
    "mode": "managed",
    "type": "aws_iam_user",
    "name": "user",
    "provider": "provider[\"registry.terraform.io/hashicorp/aws\"]",
    "instances": [
      {
        "schema_version": 0,
        "attributes": {
          "arn": "arn:aws:iam::215974853022:user/app1-svc-account",
          "force_destroy": null, #A
          "id": "app1-svc-account",
          "name": "app1-svc-account",
          "path": "/",
          "permissions_boundary": null,
          "tags": {},
          "unique_id": "AIDATESI2XGPBXYYGHJ00"
        },
        "private": "eyJzY2hlbWFdmVyc2lvbiI6IjAifQ=="
      }
    ]
  },
  ...
}
```

#A `force_destroy` is `null` instead of `true`

Why did this happen? Well, importing resources is the same as performing a `terraform refresh` on a remote resource. It reads the current state of the resource and stores it in Terraform state. The problem is that `force_destroy` isn't an AWS attribute and can't be read by making an API call. It comes from Terraform configuration, and since we haven't reconciled the state yet, it hasn't had a chance to update.

It's important to have `force_destroy` set to true because occasionally there will be a race condition between when a policy is destroyed, and when the IAM user is destroyed, causing an error. `force_destroy` deletes an IAM resource even if there are still attached policies. The easiest and best way to fix this is by performing a terraform apply, although you could also update the state manually.

```
$ terraform apply -auto-approve
...
local_file.credentials: Refreshing state... [id=4c65f8946d3bb69c819a7245fe700838e5e357fb]
module.iam["app1"].aws_iam_user.user: Modifying... [id=app1-svc-account]
module.iam["app1"].aws_iam_user.user: Modifications complete after 0s [id=app1-svc-account]

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

Now that we're back in a good state, we can cleanup as normal with a terraform destroy.

```
$ terraform destroy -auto-approve
...
module.iam["app2"].aws_iam_policy.policy[0]: Destruction complete after 1s
module.iam["app2"].aws_iam_user.user: Destruction complete after 4s
module.iam["app1"].aws_iam_user.user: Destruction complete after 4s

Destroy complete! Resources: 9 destroyed.
```

This concludes the IAM scenario. In the next section we will move on and discuss how to test infrastructure as code.

## 10.4 Testing Infrastructure as Code

Testing infrastructure as code is a bit different than testing application code. Generally speaking, when testing application code, you have at least three levels of testing:

1. **Unit Tests** – do individual parts function in isolation?
2. **Integration Tests** – do combined parts function as a component?
3. **System Tests** – does the system as a whole operate as intended?

With Terraform, we don't usually perform unit tests, as there isn't really a need to do so. Terraform configuration is mostly made up of resources and data sources, both of which are unit tested at the provider level. The closest we have to this level of testing is static analysis, which basically makes sure that configuration code is valid and has no obvious errors. Static analysis is done with either a linter, such as `terraform-lint`<sup>6</sup>, or a validation tool, such as `terraform validate`. Despite being a shallow form of testing, static analysis is still useful, because it's so quick.

---

<sup>6</sup> <https://github.com/terraform-linters/tflint>

**NOTE** Some people claim that `terraform plan` is equivalent to a “dry run”, but I disagree. `terraform plan` is not a dry run because it refreshes data sources, and data sources can execute arbitrary (potentially malicious) code.

Integration tests do make sense, as long as you are clear about what a component is. If a unit in Terraform is a single resource or data source, it follows that a component is an individual module. Modules should therefore be relatively small and encapsulated to make them easier to test

System tests (or functional tests) can be thought of as deploying an entire project, typically comprised of multiple modules and submodules. If your infrastructure deploys an application, you might also layer regression and performance testing as part of this step. We aren’t going to cover system testing in this section, because it’s subjective, and unique to the infrastructure you are deploying.

What we are going to do is write a basic integration test for a module that deploys an S3 static website. This integration test could also be generalized to work for any kind of Terraform module.

#### 10.4.1 Writing a Basic Terraform Test

HashiCorp has recently developed a Go library called `terraform-exec`<sup>7</sup> which allows for executing Terraform CLI commands programmatically. It makes it easy to write automated tests for initializing, applying and destroying Terraform configuration code. We’ll be using this library to perform integration testing on the S3 static website module.

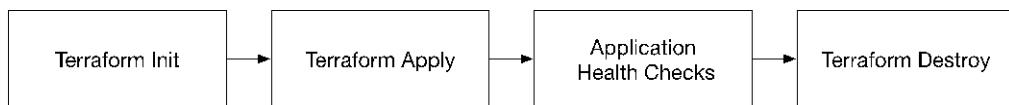


Figure 10.8 Testing a Terraform module requires calling Terraform CLI commands programmatically

#### Why not Terratest?

Terratest<sup>8</sup>, by Gruntworks, is one of the most popular Terraform testing frameworks out there. It’s been around for a number of years and has a lot of community support. Like `terraform-exec`, it’s also implemented as a Go library with helper functions for invoking Terraform CLI commands, but it’s gradually morphed into a more general-purpose testing framework. Many people use it not only for testing Terraform modules, but for Docker, K8s and Packer as well. The reason I’m not writing this section on Terratest is because there’s already a lot of material on how to use Terratest, and because there are some things that `terraform-exec` actually does better. For example, being a tool developed by HashiCorp, `terraform-exec` has feature parity with Terraform, whereas Terratest does not. You can run all Terraform CLI commands with `terraform-exec`, using any combination of flags, while Terratest only allows a small subset of the most

<sup>7</sup> <https://github.com/hashicorp/terraform-exec>

<sup>8</sup> <https://Terratest.gruntwork.io/>

common commands. Additionally, `terraform-exec` has a sister library, `terraform-json`, which allows for parsing Terraform state as regular Golang structures. This makes it easy to read anything you want from the state file. Overall, they are similar tools and could be used interchangeably, but I feel `terraform-exec` is the more polished of the two.

Listing 10.12 shows the code for a basic Terraform test. What it does is download the latest version of Terraform, initialize Terraform in a `./testfixtures` directory, perform a Terraform apply, check the health of the application, and finally cleanup with a Terraform destroy. Create a new directory in your `GOPATH` and insert the code from Listing 10.10 code into a `terraform_module_test.go` file.

#### **Listing 10.10 `terraform_module_test.go`**

```
package test

import (
    "bytes"
    "context"
    "fmt"
    "io/ioutil"
    "net/http"
    "os"
    "testing"

    "github.com/hashicorp/terraform-exec/tfexec"
    "github.com/hashicorp/terraform-exec/tfinstall"
    "github.com/rs/xid"
)

func TestTerraformModule(t *testing.T) {
    tmpDir, err := ioutil.TempDir("", "tfinstall")
    if err != nil {
        t.Error(err)
    }
    defer os.RemoveAll(tmpDir)

    latestVersion := tfinstall.LatestVersion(tmpDir, false)
    execPath, err := tfinstall.Find(latestVersion) #A
    if err != nil {
        t.Error(err)
    }

    workingDir := "./testfixtures"
    tf, err := tfexec.NewTerraform(workingDir, execPath) #B
    if err != nil {
        t.Error(err)
    }

    ctx := context.Background()
    err = tf.Init(ctx, tfexec.Upgrade(true), tfexec.LockTimeout("60s")) #C
    if err != nil {
        t.Error(err)
    }

    defer tf.Destroy(ctx) #D
}
```

```

bucketName := fmt.Sprintf("bucket_name=%s", xid.New().String())
err = tf.Apply(ctx, tfexec.Var(bucketName)) #E
if err != nil {
    t.Error(err)
}

state, err := tf.Show(context.Background())
if err != nil {
    t.Error(err)
}

endpoint := state.Values.Outputs["endpoint"].Value.(string) #F
url := fmt.Sprintf("http://%s", endpoint)
resp, err := http.Get(url)
if err != nil {
    t.Error(err)
}

buf := new(bytes.Buffer)
buf.ReadFrom(resp.Body)
t.Logf("\n%s", buf.String())

if resp.StatusCode != http.StatusOK { #G
    t.Errorf("status code did not return 200")
}
}
}

#A download latest verison of Terraform binary
#B read configuration from ./testfixtures
#C initialize Terraform
#D ensure Terraform Destroy runs even if error occurs
#E Terraform Apply with variable
#F read output value
#G fail test if status code is not 200

```

**TIP** In CI/CD, integration testing should always occur after static analysis (e.g. `terraform validate`), because integration testing takes a long time.

### 10.4.2 Test Fixtures

Before we can run the test, we need something to test against. Create a `./testfixtures` directory to hold our test fixtures. In this directory, create a new `main.tf` file with the contents of Listing 10.11. The code deploys a simple S3 static website, and outputs the URL as “`endpoint`”.

#### Listing 10.11 main.tf

```

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

variable "bucket_name" {
    type = string
}

```

```

resource "aws_s3_bucket" "website" {
  bucket = var.bucket_name
  acl = "public-read"
  policy = <<-EOF
  {
    "Version": "2008-10-17",
    "Statement": [
      {
        "Sid": "PublicReadForGetBucketObjects",
        "Effect": "Allow",
        "Principal": {
          "AWS": "*"
        },
        "Action": "s3:GetObject",
        "Resource": "arn:aws:s3:::${var.bucket_name}/*"
      }
    ]
  }
  EOF

  website {
    index_document = "index.html"
  }
}

resource "aws_s3_bucket_object" "object" {
  bucket = aws_s3_bucket.website.bucket
  key    = "index.html"
  source = "index.html"
  etag = filemd5("${path.module}/index.html") #A
  content_type = "text/html"
}

output "endpoint" { #B
  value = aws_s3_bucket.website.website_endpoint
}

```

#A the website homepage is read from a local index.html file  
#B endpoint is used by the test to check application health

We'll also need an `index.html` in the `./testfixtures` directory. This will be the website homepage. Copy the code from Listing 10.12 into `index.html`.

### **Listing 10.12 index.html**

```

<html>
<head>
  <title>Ye Olde Chocolate Shoppe</title>
</head>
<body>
  <h1>Chocolates for Any Occasion!<h1>
  <p>Come see why our chocolates are the best.</p>
</body>
</html>

```

Your working directory will now contain the following files:

```
.
└── terraform_module_test.go
    └── testfixtures
        ├── index.html
        └── main.tf

1 directory, 3 file
```

### 10.4.3 Running Tests

First import dependencies with `go mod init`:

```
$ go mod init
go: creating new go.mod: module github.com/scottwinkler/tia-chapter10
```

Then set environment variables for your AWS access and secret access keys (you could also set these as normal Terraform variables in `main.tf`).

```
$ export AWS_ACCESS_KEY_ID=<your AWS access key>
$ export AWS_SECRET_ACCESS_KEY=<your AWS secret access key>
```

**NOTE** you could also generate access keys using the IAM module from the previous section, as long as you gave it an appropriate deployment policy

We can now run the test with `go test -v`. This may take a few minutes to run, because it has to download providers, provision infrastructure, and tear it down:

```
$ go test -v
===[ RUN   TestTerraformModule
    terraform_module_test.go:63:
        <html>
        <head>
            <title>Little Princess Chocolate Shoppe</title>
        </head>
        <body>
            <h1>We Love Chocolates!</h1>
            <p>Come see why our chocolates are the best.</p>
        </body>
        </html>
--- PASS: TestTerraformModule (70.14s)
PASS
ok    github.com/scottwinkler/tia-chapter10      70.278s
```

### 10.5 Fireside Chat

Code should not only be functional; it should be readable and maintainable. This is especially true when it comes to self-service infrastructure, such as centralized repositories used by public cloud and governance teams. That being said, there is no question that refactoring

Terraform configuration is difficult. You have to be able to migrate state, anticipate runtime errors, and not lose any stateful information in the process.

Because of how hard refactoring can be, it's often a good idea to test your code at the module level. You can do this with either Terratest or the `terraform-exec` library. I recommend `terraform-exec`, because it was developed by HashiCorp, and is the more complete of the two. Ideally you should perform integration testing on all modules within your organization.

## 10.6 Summary

- `terraform taint` manually marks resources for destruction and recreation. It can be used to rotate AWS access keys, or other time sensitive resources.
- A flat module can be converted into nested modules with the help of module expansions. Module expansions permit the use of `for_each` and `count` on modules, like with resources.
- `terraform state mv` is a command that moves resources and modules around, while `terraform state rm` removes them entirely.
- Unmanaged resources can be converted to managed resources by importing them with `terraform import`. This is like performing `terraform refresh` on existing resources
- Integration tests for Terraform modules can be written by using a testing framework, such as Terratest or `terraform-exec`. A typical testing pattern is to initialize Terraform, run an `apply`, validate outputs, and `destroy`.

# 11

## *Extending Terraform by Writing Your Own Provider*

### **This chapter covers:**

- Developing a custom Terraform provider from scratch
- Implementing CRUD operations for managed resources
- Writing acceptance tests for the provider schema and resource files
- Deploying a serverless API to listen to requests from the provider
- Building and installing third-party providers

Extending Terraform by writing your own provider is one of the most satisfying things you can do with Terraform. It demonstrates a high-level proficiency with the technology and grants considerable flexibility to tweak Terraform to your needs. Although it's not as challenging as some people make it out to be, it still a huge investment in time and effort. When might it be worth it to develop your own custom provider?

Two good reasons are:

1. Wrapping an API for self-service infrastructure provisioning
2. Exposing utility functions to Terraform

Wrapping an API with a Terraform provider so that you can declare your infrastructure as code is the most common reason to create a new provider. After all, if there's not already a provider for the API you're interested in, might as well make your own (that's how all the other providers were created). Recall that Terraform is a glorified state management engine. The only way Terraform knows how to integrate with this cloud or that cloud is through providers. By developing a new provider, you are extending the reach of Terraform to manage

new API's and services. Providers make it easy for people to use an API with little to no knowledge of how the API works, so long as they can write Terraform configuration files.

The second use case, exposing utility functions to Terraform, is less conventional but still worth mentioning. Suppose, for instance, you want to perform a utility function unsupported by Terraform. This might be to fetch secrets, download build artifacts, or send an email/Slack notification at runtime. It doesn't matter what the action is, just that there's no built-in HCL function to do this for you. Since you cannot extend HCL to write your own function, you're left with creating a custom provider. The provider in question is really nothing more than a framework with which to bolt utility functions onto. Two examples of such utility providers are the Local and Archive providers. These providers don't access any remote APIs and are only useful because of the utility functions they expose to Terraform (reading and zipping files).

In this chapter, we will develop a conventional provider that interacts with a remote API to provision "Pets as Code". Pets, to clarify, are the managed Pet objects as implemented by the Petstore API. As silly as it might sound to provision "Pets as Code", the idea I want to get across is that Terraform providers can be used to provision "Anything as Code" – so long as there is an API for it. Now the idea of Minecraft and Domino's providers that were mentioned way back in the intro to chapter 1 shouldn't seem so far-fetched.

The main job of any Terraform provider is to expose resources to Terraform and initialize any shared configuration objects. Our Petstore provider will expose a single `petstore_pet` resource that can be used by Terraform to deploy and manage the lifecycle of Pets. The provider will initialize a single configuration object: the SDK client, which connects and make requests to the API. After we are finished implementing the `petstore_pet` resource, we will write a couple acceptance tests, build and install the provider, and perform end-to-end testing. Figure 11.1 depicts the end result: a Pet resource, deployed by the Petstore provider, as seen in the UI.

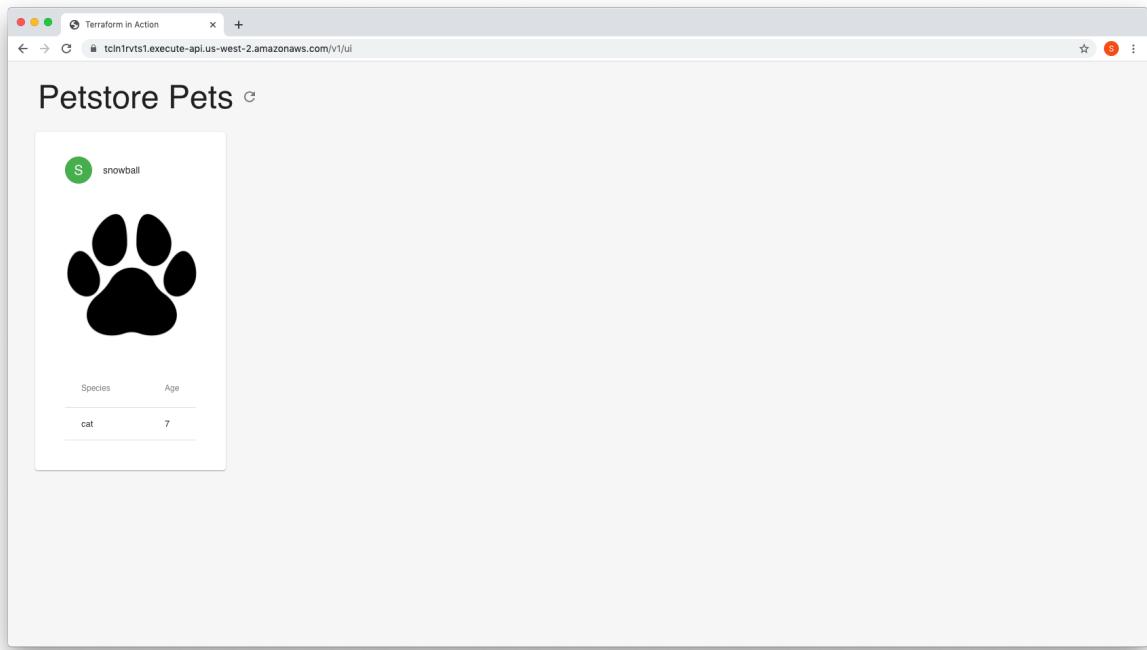


Figure 11.1 a Pet resource provisioned with the Petstore provider, as seen in the UI

## 11.1 Blueprints for a Terraform Provider

We will start by reviewing what providers are and the purpose they serve. Although we've introduced providers before – and indeed, have been using them throughout the book – we haven't explained providers from a developer's perspective. To this end, we will go through the major components that make up providers and where they fit in with the surrounding ecosystem. After that, we will go through the internal structure of the Petstore provider and the major files therein. By the end of this section you should have a big picture understanding of how providers work and be ready to begin coding the Petstore provider.

### 11.1.1 Terraform Provider Basics

Again, the main job of any Terraform provider is to expose resources to Terraform and initialize any shared configuration objects.

Resources, of course, you should already be familiar with. Resources come in two flavors: managed and unmanaged. Managed resources are just “regular” resources that implement CREATE, READ, UPDATE, DELETE (CRUD) methods for lifecycle management. Unmanaged

resources, also known as data sources or read-only resources, are less complex and only implement the `READ` part of CRUD.

*Shared configuration objects* are exactly as the name suggests: configuration objects that are shared between entities, usually for optimization or authentication purposes. These can be things like client and database connections, mutexes (concurrency locks), and temporary access keys. Terraform will always initialize these shared configuration objects first, before performing any actions against provider resources.

**NOTE** If a provider fails or “hangs” during initialization, it is almost always due to a shared configuration object having invalid or expired credentials

Shared configuration objects allude to the fact that the vast majority of providers rely on remote APIs. As a result, there are normally two prerequisites to creating your own provider:

1. **Existing API** – this one should be pretty obvious, but it’s also easily overlooked. Since Terraform makes calls against a remote API, there must be an existing remote API to make calls to. This may be your own API or someone else’s.
2. **Golang Client SDK for the API** – as providers are written in golang, you should have a golang client SDK for your API in place before proceeding. This will save you from having to make ugly, raw HTTP requests against the API

**TIP** Always have separate repositories for the client SDK and the provider! Providers are already sufficiently complicated, and there’s no need to make it harder on yourself by combining SDK code with provider code.

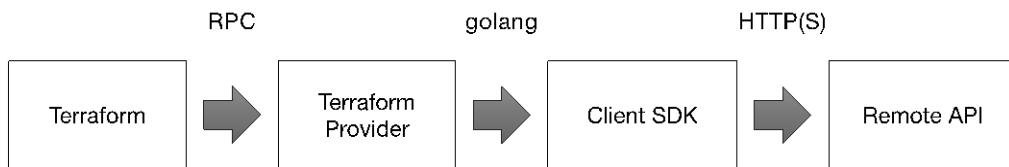


Figure 11.2 Terraform communicates with the provider over RPC, which then uses a client SDK written in golang to make HTTP requests against the remote API.

### Why Golang?

Golang is an excellent choice for open source projects because it’s fast, statically compiled, cross-platform compliant, and easy to learn. It’s no wonder then that HashiCorp chose golang for many of its major open source projects, including: Terraform, Consul, Nomad, Vault, and Packer. Despite HashiCorp’s propensity for golang, and the fact that Terraform core was written in Go, Terraform providers could theoretically be written in any language.

Providers are plugins for Terraform which communicate with Terraform over RPC (remote procedure calls). As long as providers implement the expected interface, they can be written in any language. Practically speaking, however, this is

almost never done. Providers are almost always written in Go because all the tooling and libraries for developing them is written in go. Most notably, the Terraform plugin SDK<sup>1</sup> library (formerly the `helper/*` package under Terraform core) is written exclusively in Go.

### 11.1.2 Petstore Provider Architecture

In this chapter we will develop a custom Terraform Petstore provider from scratch. This provider is relatively simple, with minimal schema configuration and exporting only a single resource, but follows all the best practices and can be used as a template for developing new providers.

There is a total of five files in the providers, these being:

1. `main.go` – the entrypoint for the provider, which is mostly uninteresting boilerplate
2. `petstore/provider.go` – where the provider definition, resource mapping, and initialization of shared configuration objects goes
3. `petstore/provider_test.go` – a file for basic acceptance tests of the provider
4. `petstore/resource_ps_pet.go` – the Pet resource which defines CRUD operations for managing a Pet resource.
5. `petstore/resource_ps_pet_test.go` – more basic acceptance tests, this time for the Pet resource

The complete file structure is therefore:

```
$ tree
.
└── main.go
    └── petstore
        ├── provider.go
        ├── provider_test.go
        ├── resource_ps_pet.go
        └── resource_ps_pet_test.go
```

**NOTE** Normally provider authors create matching read-only resources (a.k.a. data sources) to compliment managed resources, but we will not do that here to save space. I suggest consulting one of the existing Terraform providers for further examples on the subject<sup>2</sup>.

As discussed previously, we will need to have an external API to make calls against, as well as a Golang Client SDK to wrap the API. The API will be handled by a serverless Petstore app deployed on AWS, adapted from one we deployed in chapter 4. As for the SDK, we'll be using

---

<sup>1</sup> <https://github.com/hashicorp/terraform-plugin-sdk>

<sup>2</sup> <https://github.com/terraform-providers>

one that I've prepared in advance<sup>3</sup> (because creating an SDK is largely tedious and uninteresting work, no matter what people say).

The focus of this scenario is how to develop a custom provider, so I have stripped down the fluff from more comprehensive providers to just the bare bones of what it takes to constitute a real provider. Although modest, this Petstore provider adheres to the same internal structure that major cloud providers follow and is therefore a good starting point for developing your own provider.

## 11.2 Writing the Petstore Provider

In this section we will write all the functional code that goes in the Petstore provider. We'll start by setting up the entry point for the Go project, before configuring the provider schema, and finally finishing up with defining our Pet resource. By the end of this section we will have a complete provider, minus acceptance tests, which will come in the next section.

### 11.2.1 Setting up the Go Project

I will assume you have some familiarity with Go, but if you don't that's okay too. Golang is easy to understand, especially if you have any previous experience with a scripting language like Javascript, or a C-based language like Java. The first thing you need to do when getting started with Go is create a new project under your `GOPATH`. The `GOPATH` environment variable specifies the location of your Go workspace, which is where all Go code is typically kept. If no `GOPATH` is set, it is assumed to be `$HOME/go` on Unix systems and `%USERPROFILE%\go` on Windows. Under the `GOPATH` there are two subdirectories, `src` and `bin`. Create a new Go project by making an empty directory under `src` with a corresponding package directory for the Petstore provider. For example,

```
$ mkdir $GOPATH/src/github.com/scottwinkler/terraform-provider-petstore
```

**NOTE** the package directory is based on a GitHub username. For me it is `github.com/scottwinkler/terraform-provider-petstore`. In all code listings where I reference `scottwinkler`, you should replace with your own username

Next, create a file named `main.go` inside this directory containing the following Go code:

#### Listing 11.1 main.go

```
package main

import (
    "github.com/hashicorp/terraform-plugin-sdk/plugin"
    "github.com/hashicorp/terraform-plugin-sdk/terraform"
```

<sup>3</sup> <https://github.com/scottwinkler/go-petstore>

```

    "github.com/scottwinkler/terraform-provider-petstore/petstore" #A
)

func main() {
    plugin.Serve(&plugin.ServeOpts{
        ProviderFunc: func() terraform.ResourceProvider {
            return petstore.Provider()
        },
    })
}

```

#A this import needs to be relative to your project path

The `main.go` file is the main entry point for the plugin when it's invoked by Terraform. To give you an idea of what it is doing, the first line declares that this file is part of the `main` package, which is the root Go package for a given project. There are three declared imports, two from the `terraform-plugin-sdk`, and one locally referenced import. The `terraform-plugin-sdk` is a Go library created by HashiCorp to enable provider development and it has many helper functions and interfaces<sup>4</sup>. The third import may change depending on the path of your package. For me this is `github.com/scottwinkler/petstore`, for you it may be different, depending on if you wanted to put it under your own name or not (not required, but recommended).

After that comes the main function, `func main() {...}` which is the first thing called when executing the binary. All this function does is serve up the Petstore provider, which implements the `terraform.ResourceProvider` interface as defined by the plugin SDK.

### 11.2.2 Configuring the Provider Schema

Let us now consider the provider schema. The provider schema is important because it defines the attributes for the provider configuration, enumerates resources made available by the provider, and initializes any shared configuration objects. All this takes place during the `terraform init` step, when the provider is first installed.

To begin, we must first define the `Provider()` function which returns a `terraform.ResourceProvider` interface. The `ResourceProvider` interface has several mandatory fields, the one I always like to start with is `Schema`. Not to be confused with the overall provider schema, `Schema` is a parameter that outlines the allowed provider configuration attributes in Terraform. Ultimately, this will allow us to declare our provider in HCL, as shown in Snippet 11.1:

#### Snippet 11.1 provider.tf

```

provider "petstore" {
    address = var.address
}

```

---

<sup>4</sup> <https://godoc.org/github.com/hashicorp/terraform-plugin-sdk>

```
}
```

I like to start with `Schema` because the design of the provider configuration often influences the design of any resources or data sources implemented by the provider. Usually, what is passed into the provider configuration is for setting up shared configuration objects. So. things like access keys, addresses, and other shared secrets would be appropriate, whereas resource specific data would not. Our provider configuration is simple enough, as there's only a single attribute called `address` (of type `string`), which configures the endpoint of the Petstore server. Note that the Petstore API is unauthenticated, hence there is no need for shared secrets.

**WARNING!** For any production API you should always implement authentication for your API and never bake secrets into the provider source code. Do not follow my example!

One more thing to mention about `address` is that we may wish to optionally set it with an environment variable, rather than with a Terraform variable, so that the provider can be run in automation. We can do with the help of a pre-built function from the plugin SDK, called `schema.EnvDefaultFunc`. This function makes it possible to set a default environment variable to use if the attribute is not directly set in the provider configuration.

**TIP** it is good idea to make critical configuration attributes, such as access keys and addresses, optionally configurable as environment variables – for ease in automation

Go ahead and create a folder within your project called “petstore”, and in this folder create a `provider.go` file with the following code from Listing 11.2.

**NOTE** the `petstore` folder is officially known as a golang package, and will contain all provider code, besides the main function.

### Listing 11.2 provider.go

```
package petstore

import (
    "net/url"

    "github.com/hashicorp/terraform-plugin-sdk/helper/schema"
    "github.com/hashicorp/terraform-plugin-sdk/terraform"
    sdk "github.com/scottwinkler/go-petstore"
)

func Provider() terraform.ResourceProvider {
    return &schema.Provider{
        Schema: map[string]*schema.Schema{
            "address": &schema.Schema{
                Type:     schema.TypeString,
                Optional: true,
            }
        }
}
```

```

        DefaultFunc: schema.EnvDefaultFunc("PETSTORE_ADDRESS", nil),
    },
}
}
```

Now that we have the basic provider schema out of the way, we must register all the Resources that the provider makes available to Terraform in a map structure. The keys of the map will be the name of the resource, and the value of the map will be a pointer to `schema.Resource` objects. This map will have only a single resource, called `petstore_pet`, which will be used to manage the lifecycle of a Pet entity. We have not created this resource yet, but let us preemptively add a function called `resourcePSPet()` that we define in the next section. Edit the `provider.go` file to add this resource map.

### **Listing 11.3 provider.go**

```

package petstore

import (
    "net/url"

    "github.com/hashicorp/terraform-plugin-sdk/helper/schema"
    "github.com/hashicorp/terraform-plugin-sdk/terraform"
    sdk "github.com/scottwinkler/go-petstore"
)

func Provider() terraform.ResourceProvider {
    return &schema.Provider{
        Schema: map[string]*schema.Schema{
            "address": &schema.Schema{
                Type:      schema.TypeString,
                Optional: true,
                DefaultFunc: schema.EnvDefaultFunc("PETSTORE_ADDRESS", nil),
            },
            ResourcesMap: map[string]*schema.Resource{
                "petstore_pet": resourcePSPet(),
            },
        },
    }
}
```

Lastly, we need to initialize any shared configuration objects. For our purposes, this will just be the client that the SDK uses to make API requests against the Petstore server. The logic for doing this is encapsulated in the `ConfigureFunc` field of the provider schema. `ConfigureFunc` is a function which accepts only one argument (`*schema.ResourceData`), and produces two outputs (`interface{}` and `error`). The argument `*schema.ResourceData` is a pointer to a structure which contains metadata about the provider itself. Note that the first output, `interface{}` is just a way to stipulate generics in Go, and is a placeholder for anything. This generic output is passed to all resources during CRUD operations, and is therefore a convenient container to store shared configuration objects.

The complete code for `provider.go` is shown in Listing 11.4:

#### **Listing 11.4 provider.go**

```
package petstore

import (
    "net/url"

    "github.com/hashicorp/terraform-plugin-sdk/helper/schema"
    "github.com/hashicorp/terraform-plugin-sdk/terraform"
    sdk "github.com/scottwinkler/go-petstore"
)

func Provider() terraform.ResourceProvider {
    return &schema.Provider{
        Schema: map[string]*schema.Schema{
            "address": &schema.Schema{
                Type:           schema.TypeString,
                Optional:      true,
                DefaultFunc:   schema.EnvDefaultFunc("PETSTORE_ADDRESS", nil),
            },
            ResourcesMap: map[string]*schema.Resource{
                "petstore_pet": resourcePSPet(),
            },
            ConfigureFunc: providerConfigure,
        }
    }
}

func providerConfigure(d *schema.ResourceData) (interface{}, error) {
    hostname, _ := d.Get("address").(string)
    address, _ := url.Parse(hostname)
    cfg := &sdk.Config{
        Address: address.String(),
    }
    return sdk.NewClient(cfg)
}
```

As a side note, the `terraform providers schema` command can be used to print detailed schemas for the providers used in the current configuration. An example of the Petstore provider schema (after it is built and installed) is shown in Snippet 11.2.

#### **Snippet 11.2 terraform providers schema**

```
$ terraform providers schema
{"format_version":"0.1","provider_schemas":{"petstore":{"provider":{"version":0,"block":{"attributes":{"address":{"type":"string","optional":true}}},"resource_schemas":{"petstore_pet":{"version":0,"block":{"attributes":{"age":{"type":"number","required":true}, "id":{"type":"string","optional":true,"computed":true}, "name":{"type":"string","required":true}, "species":{"type":"string","required":true}}}}}}}}
```

## 11.3 Defining a Pet Resource

The function `resourcePSPet()` has a method signature accepting zero input arguments and returning a single output, a `schema.Resource` interface. Our Pet resource is an implementation of this `schema.Resource` interface. As you might have guessed, four of the fields on this interface have to do with function hooks invoked during CRUD lifecycle management, these being:

- `Create` – a pointer to a function that's invoked when a create lifecycle event is triggered. Create lifecycle events are triggered when new resources are created, such as during an initial apply and during force new updates.
- `Read` – a pointer to a function that's invoked when a read lifecycle event is triggered. Read events are triggered during the generation of an execution plan, so as to determine if configuration drift has occurred. Additionally, the `Read()` function is also usually called as a side effect of `Create()` and `Update()`.
- `Update` – a pointer to a function that's invoked when an update lifecycle event is triggered. It handles in-place (a.k.a. non-destructive) updates. This field may be omitted if all attributes in the resource schema are marked as `ForceNew`.
- `Delete` – a pointer to a function that's invoked when a delete lifecycle event is triggered. Delete lifecycle events are triggered: a) during a `terraform destroy`, b) when a resource is removed from configuration (or marked as tainted), followed by a `terraform apply`, and c) when an attribute marked as `ForceNew` has been changed

It's important to know when each of the four CRUD functions will be invoked, to be able to predict and handle any errors that may occur. During an initial apply with no previous state, `Create()` is called by Terraform, which has the side effect of calling `Read()`. During a `terraform plan`, `Read()` is called all by itself. During an in-place update, `Read()` is called first, like during the plan, and then `Update()` is called, which has the side effect of calling `Read()` again. Force new updates call `Read()`, then `Delete()`, then `Create()`, and finally `Read()` again. Destroy operations always call `Read()` and then `Delete()`. A visual reference diagram is shown in figure 11.3.

Step #	Command	Invoked Functions
1	terraform apply (initial deploy)	Create() Read()
2	terraform plan	Read()
3	terraform apply (update)	Read() Update() Read()
4	terraform apply (force new update)	Read() Delete() Create() Read()
5	terraform destroy	Read() Delete()

**Figure 11.3** different methods are invoked based on the command given, as well as the current state and configuration. It's a bit complicated because some methods (create and update) have the side effect of calling other methods (read).

Besides CRUD methods, the resource schema also has another required field called `Schema`. Like the provider schema, this is a map of attributes that the resource defines. The type of each attribute must be specified, as well as whether the attribute is required, optional or force new. We will have three attributes for our Pet resource: `name`, `species` and `age`. `Name` will be optional because not all pets have to have names. `Species` will be marked as required and force new (because making a change to a pet's species is kind of a big deal). `Age` is an integer type used that's required, but not marked as `ForceNew`, because it's highly likely to be incremented in the future.

Let us now define the function for the Pet resource in a separate file called `resource_ps_pet.go`.

#### Listing 11.5 `resource_ps_pet.go`

```
package petstore

import (
    "github.com/hashicorp/terraform-plugin-sdk/helper/schema"
    sdk "github.com/scottwinkler/go-petstore"
)

func resourcePSPet() *schema.Resource {
    return &schema.Resource{
        Create: resourcePSPetCreate,
```

```
Read: resourcePSPetRead,
Update: resourcePSPetUpdate,
Delete: resourcePSPetDelete,

Schema: map[string]*schema.Schema{
    "name": {
        Type:     schema.TypeString,
        Optional: true,
        Default:  "",
    },
    "species": {
        Type:     schema.TypeString,
        ForceNew: true,
        Required: true,
    },
    "age": {
        Type:     schema.TypeInt,
        Required: true,
    },
},
```

Next, we will define the Create(), Read(), Update, and Delete() methods.

### 11.3.1 Defining Create() – CRUD Lifecycle Management

`Create()` is a function responsible for provisioning a new resource based on user supplied input, and setting the unique ID of the resource. ID is important because without it, the resource won't be marked as "created" by Terraform, and neither will it be persisted to the state file. The implementation of `Create()` typically means performing a POST request against the target API, waiting for an API response, handling any retry logic, and invoking a `Read()` operation afterwards.

**TIP** although you could write the logic for performing a raw HTTP POST request inline in the create function, I would not recommend it. That's what the client SDK is for.

Because we already have a Petstore client SDK (which encapsulates much of the tedious logic of interacting with the API), the `Create()` method becomes incredibly simple, as shown in Listing 11.6.

## **Listing 11.6 resource ps pet.go**

```
package petstore

import (
    "github.com/hashicorp/terraform-plugin-sdk/helper/schema"
    sdk "github.com/scottwinkler/go-petstore"
)

func resourcePSPet() *schema.Resource {
    return &schema.Resource{
        Create: resourcePSPetCreate,
```

```

Read: resourcePSPetRead,
Update: resourcePSPetUpdate,
Delete: resourcePSPetDelete,

Schema: map[string]*schema.Schema{
    "name": {
        Type:     schema.TypeString,
        Optional: true,
        Default:  "",
    },
    "species": {
        Type:     schema.TypeString,
        ForceNew: true,
        Required: true,
    },
    "age": {
        Type:     schema.TypeInt,
        Required: true,
    },
},
}
}

func resourcePSPetCreate(d *schema.ResourceData, meta interface{}) error {
    conn := meta.(*sdk.Client) #A
    options := sdk.PetCreateOptions{
        Name:     d.Get("name").(string),
        Species: d.Get("species").(string),
        Age:      d.Get("age").(int),
    }

    pet, err := conn.Pets.Create(options) #B
    if err != nil {
        return err
    }

    d.SetId(pet.ID) #C
    resourcePSPetRead(d, meta) #D
    return nil #E
}
}

```

#A the meta object comes from the provider configuration. In this case, it happens to be the instantiated client SDK, which is why it is cast as such.  
#B the SDK handles making the POST request with a request object  
#C the resource ID is set based on the ID field of the response object  
#D best practice is to call Read() as a side effect of Create()  
#E returning nil means no errors have occurred

### 11.3.2 Defining Read() – CRUD Lifecycle Management

You can think of `Read()` as a non-destructive operation that checks what the remote (or actual) state of the resource is. It is called whenever an execution plan would be generated and as a side effect of both `Update()` and `Create()`. Generally, `Read()` uses the unique resource ID to perform a lookup against the API, although it can also use other attribute that can be used to uniquely identify the resource. Regardless of how the lookup is done, the

response from the API is considered authoritative. If the actual state doesn't match the desired state as described in the current configuration/state file, an update will be triggered during the next apply.

**WARNING!** `Read()` should always return the same resource from the API. If it does not, you will end up with orphaned resources. *Orphaned resources* are resources originally created by Terraform, which have been lost track of, and are now considered unmanaged.

Please add the code from Listing 11.7 to the bottom of the `resource_ps_pet.go` file to implement `Read()`. All this does is use the Petstore SDK to perform a lookup of the Pet resource based on ID, throw an error if one has occurred, and set the attributes based on the response from the API.

### Listing 11.7 `resource_ps_pet.go`

```
...
func resourcePSPetCreate(d *schema.ResourceData, meta interface{}) error {
    conn := meta.(*sdk.Client)
    options := sdk.PetCreateOptions{
        Name:      d.Get("name").(string),
        Species:   d.Get("species").(string),
        Age:       d.Get("age").(int),
    }

    pet, err := conn.Pets.Create(options)
    if err != nil {
        return err
    }

    d.SetId(pet.ID)
    resourcePSPetRead(d, meta)
    return nil
}

func resourcePSPetRead(d *schema.ResourceData, meta interface{}) error {
    conn := meta.(*sdk.Client)
    pet, err := conn.Pets.Read(d.Id()) #A
    if err != nil {
        return err
    }
    d.Set("name", pet.Name) #B
    d.Set("species", pet.Species) #B
    d.Set("age", pet.Age) #B
    return nil
}
```

#A Using the Petstore SDK to perform a lookup of the Pet resource, based on ID

#B Setting the attributes based on the output of the response. If there is a discrepancy between the response object and the current configuration then Terraform will throw an error

### 11.3.3 Defining Update() – CRUD Lifecycle Management

Although Terraform is often touted as an immutable infrastructure as code technology (and indeed I described it as such in chapter 1), strictly speaking, it's not. Almost all resources that Terraform manages are mutable to some degree. As a reminder, *immutable infrastructure* is the concept of never performing updates in-place. If an update needs to occur, it takes place by tearing down the old infrastructure (such as a server) and replacing it with new infrastructure pre-configured to the desired state. By contrast, *mutable infrastructure* is when existing resources are allowed to persist through in-place updates or patches instead of deleting and recreating them. Only if every attribute on a resource were marked as `ForceNew` (and almost no resource is this way), could it be described as immutable.

The purpose of `Update()` is to perform non-destructive, in-place updates on existing resources. It's a tricky method to implement, and it may be tempting to skip the need for it entirely by marking all attributes as `ForceNew`, but I wouldn't recommend doing this. Force new updates are inconvenient from a user perspective because it takes longer for changes to propagate. This is an example where a good user experience matters more than ease of development or strict adherence to infrastructure immutability.

The sole responsibilities of `Update()` is to check if changes have occurred to any mutable attributes and act accordingly. Typically, this means performing a HTTP `PATCH` request with the changes in a payload object, followed by a `GET` request to ensure the managed resource has been updated correctly. Again, we will rely on the Petstore client SDK to abstract away the uninteresting tidbits of API logic.

---

#### **Creating a Client SDK for an API**

A software development kit (SDK) is a collection of libraries, tools, documentation, and example code used by developers to create applications for specific platforms. An SDK for an API (a.k.a. client SDK or client library) is a set of reusable functions used to interface with the API in a particular programming language. It authenticates to the server, makes HTTP requests, processes responses, and handles any errors that may occur. You can choose to either lovingly create such a library from scratch, or generate one from a specification file, but the goal of any good SDK should be to make it easy for users to invoke the API.

An SDK should always be written against an API specification file. There are many kinds of API specifications, but the one most common one for RESTful API's is the OpenAPI specification<sup>5</sup> (formerly known as Swagger). The OpenAPI specification is an API description format that allows you to describe the inputs and outputs of REST APIs in YAML or JSON. Good practice is to write the API specification first, then write the SDK and/or API to meet that specification. One interesting possibility that writing your API to a specification brings is being able to generate server stubs (API implementation files) and client libraries on the fly. Both save developer time and make it easy to support additional programming languages. Nevertheless, generated code is not always a perfect fit, and you may be better off writing custom code instead. For example, if you intend for your API to only be called by a Terraform provider, I suggest writing

---

<sup>5</sup> <https://github.com/OAI/OpenAPI-Specification/blob/master/IMPLEMENTATIONS.md#implementations>

the golang client library from scratch. It may be boring and tedious work, but at least you can tailor the library for exactly how it will be used by the provider.

Add in the code from Listing 11.7 to the bottom of `resource_ps_pet.go` to define and implement `Update()`.

#### **Listing 11.8 resource\_ps\_pet.go**

```
...
func resourcePSPetRead(d *schema.ResourceData, meta interface{}) error {
    conn := meta.(*sdk.Client)
    pet, err := conn.Pets.Read(d.Id())
    if err != nil {
        return err
    }
    d.Set("name", pet.Name)
    d.Set("species", pet.Species)
    d.Set("age", pet.Age)
    return nil
}

func resourcePSPetUpdate(d *schema.ResourceData, meta interface{}) error {
    conn := meta.(*sdk.Client)
    options := sdk.PetUpdateOptions{}
    if d.HasChange("name") { #A
        options.Name = d.Get("name").(string)
    }
    if d.HasChange("age") { #A
        options.Age = d.Get("age").(int)
    }
    conn.Pets.Update(d.Id(), options)
    return resourcePSPetRead(d, meta) #B
}
```

#A `Update()` needs to check each non-`ForceNew` attribute to see if it has changed  
#B Like `Create()`, `Update()` needs to call `Read()` as a side effect

#### **11.3.4 Defining Delete() – CRUD Lifecycle Management**

The last lifecycle method to implement is `Delete()`. This method is responsible for making an API request to delete the existing resource and set the resource ID to `nil` (which marks the resource as destroyed and removes it from the state file). I always find `Delete()` to be the easiest method implement, but it's still important not to make any mistakes. If `Delete()` fails to delete (such as if the API experienced an internal error due to poor implementation), you will be left with orphaned resources.

**NOTE** You can call `Read()` after `Delete()` if you wish to ensure that a resource has actually been destroyed, but normally this is not done. `Delete()` is presumed to succeed if the response from the server says that it has succeeded. Server errors should be handled by the server or SDK.

The code for `Delete()` is shown in Listing 11.9.

### **Listing 11.9 resource\_ps\_pet.go**

```
...
func resourcePSPetUpdate(d *schema.ResourceData, meta interface{}) error {
    conn := meta.(*sdk.Client)
    options := sdk.PetUpdateOptions{}
    if d.HasChange("name") {
        options.Name = d.Get("name").(string)
    }
    if d.HasChange("age") {
        options.Age = d.Get("age").(int)
    }
    conn.Pets.Update(d.Id(), options)
    return resourcePSPetRead(d, meta)
}

func resourcePSPetDelete(d *schema.ResourceData, meta interface{}) error {
    conn := meta.(*sdk.Client)
    conn.Pets.Delete(d.Id())
    return nil
}
```

For your reference, the complete code of `resource_ps_pet.go` is presented in Listing 11.10.

### **Listing 11.10 resource\_ps\_pet.go**

```
package petstore

import (
    "github.com/hashicorp/terraform-plugin-sdk/helper/schema"
    sdk "github.com/scottwinkler/go-petstore"
)

func resourcePSPet() *schema.Resource {
    return &schema.Resource{
        Create: resourcePSPetCreate,
        Read:   resourcePSPetRead,
        Update: resourcePSPetUpdate,
        Delete: resourcePSPetDelete,

        Schema: map[string]*schema.Schema{
            "name": {
                Type:     schema.TypeString,
                Optional: true,
                Default:  "",
            },
            "species": {
                Type:     schema.TypeString,
                ForceNew: true,
                Required: true,
            },
            "age": {
                Type:     schema.TypeInt,
                Required: true,
            },
        },
    }
}
```

```

        },
    }
}

func resourcePSPetCreate(d *schema.ResourceData, meta interface{}) error {
    conn := meta.(*sdk.Client)
    options := sdk.PetCreateOptions{
        Name:    d.Get("name").(string),
        Species: d.Get("species").(string),
        Age:     d.Get("age").(int),
    }

    pet, err := conn.Pets.Create(options)
    if err != nil {
        return err
    }

    d.SetId(pet.ID)
    resourcePSPetRead(d, meta)
    return nil
}

func resourcePSPetRead(d *schema.ResourceData, meta interface{}) error {
    conn := meta.(*sdk.Client)
    pet, err := conn.Pets.Read(d.Id())
    if err != nil {
        return err
    }
    d.Set("name", pet.Name)
    d.Set("species", pet.Species)
    d.Set("age", pet.Age)
    return nil
}

func resourcePSPetUpdate(d *schema.ResourceData, meta interface{}) error {
    conn := meta.(*sdk.Client)
    options := sdk.PetUpdateOptions{}
    if d.HasChange("name") {
        options.Name = d.Get("name").(string)
    }
    if d.HasChange("age") {
        options.Age = d.Get("age").(int)
    }
    conn.Pets.Update(d.Id(), options)
    return resourcePSPetRead(d, meta)
}

func resourcePSPetDelete(d *schema.ResourceData, meta interface{}) error {
    conn := meta.(*sdk.Client)
    conn.Pets.Delete(d.Id())
    return nil
}

```

## 11.4 Writing Acceptance Tests

A provider isn't complete until it's been thoroughly tested. Tests are important because they give you the confidence to know that your code is relatively bug free and working as intended.

Writing good tests can be tough, but well it's worth the effort, especially on large code bases with multiple contributors. In this section we will write two test files, one for the provider schema and one for the Pet resource.

**NOTE** If you wish to contribute to any open source provider, you will be expected to include test cases along with your pull request.

### 11.4.1 Testing the Provider Schema

The primary purpose of testing the provider schema is to ensure that the provider:

1. Can be successfully initialized
2. Has a valid internal schema
3. Has all environment variables required for testing set

**NOTE** Sometimes people will also test the individual attributes of the provider, along with various ways to configure the provider

Please create a `provider_test.go` file with the following code:

#### Listing 11.11 provider\_test.go

```
package petstore

import (
    "os"
    "testing"

    "github.com/hashicorp/terraform-plugin-sdk/helper/schema"
    "github.com/hashicorp/terraform-plugin-sdk/terraform"
)

var testAccProviders map[string]terraform.ResourceProvider
var testAccProvider *schema.Provider

func init() { #A
    testAccProvider = Provider().(*schema.Provider)
    testAccProviders = map[string]terraform.ResourceProvider{
        "petstore": testAccProvider,
    }
}

func TestProvider(t *testing.T) { #B
    if err := Provider().(*schema.Provider).InternalValidate(); err != nil {
        t.Fatalf("err: %s", err)
    }
}

func TestProvider_impl(t *testing.T) { #C
    var _ terraform.ResourceProvider = Provider()
}

func testAccPreCheck(t *testing.T) { #D}
```

```

if os.Getenv("PETSTORE_ADDRESS") == "" {
    t.Fatal("PETSTORE_ADDRESS must be set for acceptance tests")
}

err := testAccProvider.Configure(terraform.NewResourceConfigRaw(nil))
if err != nil {
    t.Fatal(err)
}
}

#A initializes global variables
#B tests that the provider schema is valid
#C tests that the provider can be initialized
#D tests that PETSTORE_ADDRESS environment variable is set

```

### 11.4.2 Testing the Pet Resource

Writing a test for a Terraform resource is more nuanced than writing basic tests for the provider schema because it requires learning a custom testing framework developed by HashiCorp. Don't worry though, you don't have to know much about this testing framework for the scenario. The framework in question is worth looking into, however, because it allows you to do cool stuff like run sequences of tests against resources with various configurations and run pre-processor and post-processor functions. It was tailor for testing Terraform resources and is certainly easier than writing your own testing framework.

There's a lot you can do with the testing framework, and I recommend looking at examples from open source providers to see what other people have done. At a bare minimum, however, a resource test file requires the following:

- Basic create/destroy test with validation that attributes get set in the state file
- A function to check that all test resources have been destroyed
- Test HCL configuration with all input attributes set

Test code for the Pet resource is shown in Listing 11.12. Please copy this into a `resource_ps_pet_test.go` file, under the `petstore` directory.

#### **Listing 11.12 `resource_ps_pet_test.go`**

```

package petstore

import (
    "fmt"
    "testing"

    "github.com/hashicorp/terraform-plugin-sdk/helper/resource"
    "github.com/hashicorp/terraform-plugin-sdk/terraform"
    sdk "github.com/scottwinkler/go-petstore"
)

func TestAccPSPet_basic(t *testing.T) { #A
    resourceName := "petstore_pet.pet"

    resource.Test(t, resource.TestCase{

```

```

PreCheck:    func() { testAccPreCheck(t) }, #B
Providers:   testAccProviders, #C
CheckDestroy: testAccCheckPSPetDestroy, #D
Steps: []resource.TestStep{ #E
    {
        Config: testAccPSPetConfig_basic(),
        Check: resource.ComposeTestCheckFunc(
            resource.TestCheckResourceAttr(resourceName, "name",
                "Winston"),
            resource.TestCheckResourceAttr(resourceName, "species",
                "cat"),
            resource.TestCheckResourceAttr(resourceName, "age", "2"),
        ),
    },
},
}

func testAccCheckPSPetDestroy(s *terraform.State) error {
    for _, rs := range s.RootModule().Resources {
        if rs.Type != "petstore_pet" {
            continue
        }
        if rs.Primary.ID == "" {
            return fmt.Errorf("No instance ID is set")
        }
        conn := testAccProvider.Meta().(*sdk.Client)
        pet, err := conn.Pets.Read(rs.Primary.ID)
        if err != sdk.ErrResourceNotFound {
            return fmt.Errorf("Pet %s still exists", pet.ID)
        }
    }
    return nil
}

func testAccPSPetConfig_basic() string {
    return fmt.Sprintf(`resource "petstore_pet" "pet" {
    name = "Winston"
    species = "cat"
    age = 2
}`)
}
}

```

#A The most basic acceptance test for a resource

#B PreCheck ensures that the PETSTORE\_ADDRESS attribute has been set

#C Use the global provider that was initialized earlier with the init() function

#D Sweeper function to ensure the resource has been destroyed

#E Simple test which creates a resource using sample configuration and checks that the set attributes are as expected

## 11.5 Build, Test, Deploy

The code for the provider is now complete, but we still have a few tasks left to do. First, we need an actual Petstore API to test against, then we will need to test and build the provider binary, and finally we need to run end-to-end tests with real configuration code.

### 11.5.1 Deploy the Petstore API

For your convenience, I have packaged the API into a module that can be easily deployed with a few lines of Terraform code. This module deploys a serverless backend with an API gateway, a Lambda function and an RDS database. It parallels the architecture of the serverless app deployed in Chapter 5, except for AWS. Basically, I took the webapp deployed in chapter 4, simplified it somewhat, and tweaked it to run on serverless technologies.

Below is the code for the Petstore module. Create a new, separate Terraform workspace with this file.

**Listing 11.13 petstore.tf**

```
terraform {
  required_version = "~> 0.12"
  required_providers {
    aws = "~> 2.45"
    random = "~> 2.2"
  }
}

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

module "petstore" {
  source  = "scottwinkler/petstore/aws"
}

output "address" {
  value = module.petstore.address
}
```

Deploy as per normal, by performing a `terraform init`, followed by a `terraform apply`.

```
$ terraform init
...
Terraform has been successfully initialized!

$ terraform apply
...
Plan: 24 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

Enter a value:
```

After confirming the apply, it should take about 5-10 minutes to deploy the serverless application. At the end, you will get the address of your deployed API. Copy this down, we will need it in the next section.

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

```
Outputs:
```

```
address = https://tcln1rvts1.execute-api.us-west-2.amazonaws.com/v1
```

If you navigate to this address in the browser it will redirect you to a simple web UI. Notice that it is empty to start with, because there are no Pets in the database yet.

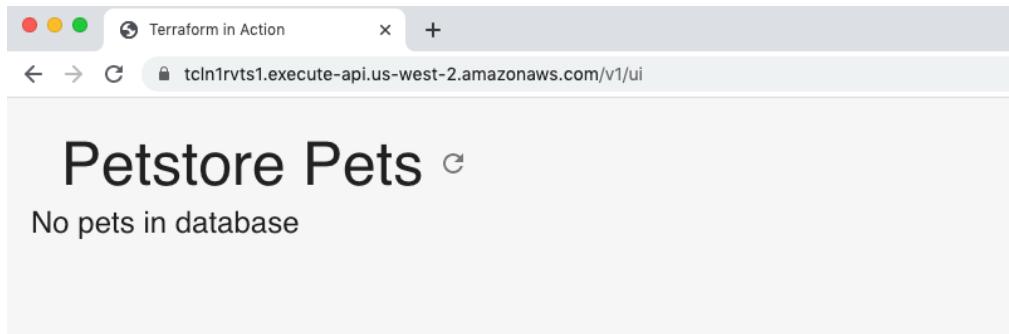


Figure 11.4 Initially there are no pets in the database, so the web UI doesn't show anything

### 11.5.2 Test and Build the Provider

Before running acceptance tests, you must set the `TF_ACC` environment variable to 1:

```
$ export TF_ACC=1
```

Now run the acceptance tests with: `go test -v ./...`

```
$ go test -v ./...
?     github.com/scottwinkler/terraform-provider-petstore [no test files]
?     github.com/scottwinkler/terraform-provider-petstore/example [no test files]
==== RUN  TestProvider
--- PASS: TestProvider (0.00s)
==== RUN  TestProvider_implementation
--- PASS: TestProvider_implementation (0.00s)
==== RUN  TestAccPSPet_basic
--- FAIL: TestAccPSPet_basic (0.00s)
    provider_test.go:33: PETSTORE_ADDRESS must be set for acceptance tests
FAIL
FAIL    github.com/scottwinkler/terraform-provider-petstore/petstore  1.385s
FAIL
```

As you can see, we need to set the `PETSTORE_ADDRESS` environment variable before proceeding, due to the `PreCheck` function in `TestAccPSPet_basic()`. Do that now, then re-run all tests:

```
$ export PETSTORE_ADDRESS=<your Petstore address>
$ go test -v ./...
?     github.com/scottwinkler/terraform-provider-petstore [no test files]
```

```
?      github.com/scottwinkler/terraform-provider-petstore/example [no test files]
==> RUN  TestProvider
--- PASS: TestProvider (0.00s)
==> RUN  TestProvider_implementation
--- PASS: TestProvider_implementation (0.00s)
==> RUN  TestAccPreCheck
--- PASS: TestAccPreCheck (0.00s)
==> RUN  TestAccPSPet_basic
--- PASS: TestAccPSPet_basic (5.59s)
PASS
ok    github.com/scottwinkler/terraform-provider-petstore/petstore 6.223s
```

All tests should now pass, so the provider is ready to be built. You can do that with a `go mod init` followed by a `go build`.

```
$ go mod init
go: creating new go.mod: module github.com/scottwinkler/terraform-provider-petstore
$ go build
go: finding github.com/hashicorp/terraform-plugin-sdk v1.5.0
go: downloading github.com/hashicorp/terraform-plugin-sdk v1.5.0
go: extracting github.com/hashicorp/terraform-plugin-sdk v1.5.0
```

The binary file is built for your current architecture and operating systems and placed in the local working directory.

```
$ ls -o
total 56976
-rw-r--r-- 1 swinkler 216 Jan 20 19:56 go.mod
-rw-r--r-- 1 swinkler 45873 Jan 20 19:56 go.sum
-rw-r--r-- 1 swinkler 337 Jan 20 21:20 main.go
drwxr-xr-x 6 swinkler 192 Jan 20 21:21 petstore
-rwxr-xr-x 1 swinkler 29108564 Jan 20 22:26 terraform-provider-petstore
```

**LSTIP** Most provider authors use a `Makefile` and CI triggers to automate the steps of building, testing, and distributing the provider. I recommend looking at some simpler providers, like `terraform-provider-null` and `terraform-provider-tfe` for inspiration

Move the binary into a new `examples` directory with the following commands:

```
$ mkdir examples
$ mv terraform-provider-petstore ./examples
```

Switch into the `examples` directory and create a `main.tf` to test the provider. The code is shown in Listing 11.14.

#### Listing 11.14 main.tf

```
provider "petstore" {
  address = "https://tcln1rvts1.execute-api.us-west-2.amazonaws.com/v1"#A
}

resource "petstore_pet" "my_pet" {
  name = "snowball"
  species = "cat"
```

```
    age = 7
}
```

#A Your provider address goes here

If we were to initialize the provider now with `TF_LOG` set to `TRACE`, we would see that Terraform identifies the binary file as a “legacy provider” with a version of 0.0.0. The reason for this is because the name of the provider binary tells Terraform what version it actually is.

```
Rename the binary file from terraform-provider-petstore to terraform-provider-
petstore_v1.0.0 and perform a terraform init.
$ mv terraform-provider-petstore terraform-provider-petstore_v1.0.0
$ terraform init
```

Initializing the backend...

Initializing provider plugins...

The following providers do not have any version constraints in configuration,
so the latest version was installed.

To prevent automatic upgrades to new major versions that may contain breaking
changes, it is recommended to add `version = "..."` constraints to the
corresponding provider blocks in configuration, with the constraint strings
suggested below.

```
* provider.petstore: version = "~> 1.0"
```

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running `"terraform plan"` to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.

Now that Terraform has detected the provider version and installed it successfully, run an
apply in the workspace:

```
$ terraform apply
```

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
+ create

Terraform will perform the following actions:

```
# petstore_pet.my_pet will be created
+ resource "petstore_pet" "my_pet" {
    + age      = 7
    + id       = (known after apply)
    + name     = "snowball"
    + species  = "cat"
}
```

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

```
Do you want to perform these actions?
```

```
Terraform will perform the actions described above.
```

```
Only 'yes' will be accepted to approve.
```

```
Enter a value:
```

**NOTE** it can take up to 30 seconds for an API request succeed, due to the serverless nature of the API. Once it has warmed up, request time will be much faster.

As you can see, Terraform recognizes our provider as valid and wants to create a new Pet resource! Confirm the apply to proceed.

```
petstore_pet.my_pet: Creating...
petstore_pet.my_pet: Still creating... [10s elapsed]
petstore_pet.my_pet: Creation complete after 11s [id=d2c984f6-f892-4f4e-aacb-483ebd90747d]
```

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

The resource now exists as a record in the Petstore database. You can view it by navigating to the UI again and seeing that a new resource exists.

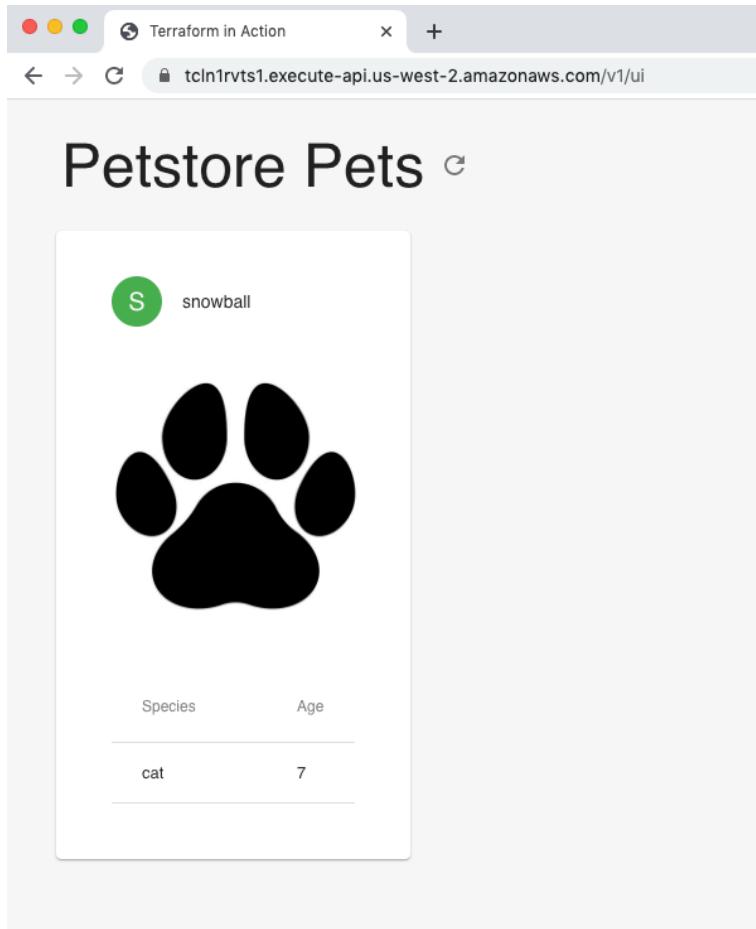


Figure 11.5 the provisioned Pet resource, as seen in the UI

**NOTE** another way to verify the resource exists is by querying the raw API e.g. a GET against <https://tcln1rvts1.execute-api.us-west-2.amazonaws.com/v1/api/pets>

The resource has been recorded in the state file which we can view with a `terraform state show`.

```
$ terraform state show petstore_pet.my_pet
# petstore_pet.my_pet:
resource "petstore_pet" "my_pet" {
    age      = 7
    id       = "d2c984f6-f892-4f4e-aacb-483ebd90747d"
    name     = "snowball"
    species  = "cat"
}
```

If we make changes to the configuration code, for example incrementing age from 7 to 8, we would get the following message during the next apply:

```
$ terraform apply
petstore_pet.my_pet: Refreshing state... [id=5b1c5fa5-e27b-466d-9f5d-05842451e6e3]

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  ~ update in-place

Terraform will perform the following actions:

# petstore_pet.my_pet will be updated in-place
~ resource "petstore_pet" "my_pet" {
    ~ age      = 7 -> 8
    id        = "5b1c5fa5-e27b-466d-9f5d-05842451e6e3"
    name      = "snowball"
    species   = "cat"
}

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

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

Enter a value:
```

Let us now clean up by deleting the resource from the API with a `terraform destroy`.

```
$ terraform destroy -auto-approve
petstore_pet.my_pet: Refreshing state... [id=d2c984f6-f892-4f4e-aacb-483ebd90747d]
petstore_pet.my_pet: Destroying... [id=d2c984f6-f892-4f4e-aacb-483ebd90747d]
petstore_pet.my_pet: Destruction complete after 1s
Destroy complete! Resources: 1 destroyed.
```

**NOTE** It is helpful to set `TF_LOG=TRACE` when testing providers, to ensure that API requests and responses are as expected

This concludes the scenario. Don't forget to tear down the Petstore API with a `terraform destroy` as well!

**NOTE** as a side effect of improved VPC networking changes to AWS Lambda, it can take up to 45 minutes to delete a Lambda function with a VPC configuration (which this module has). AWS is working to resolve the issue<sup>6</sup>.

---

<sup>6</sup> <https://aws.amazon.com/blogs/compute/announcing-improved-vpc-networking-for-aws-lambda-functions/>

## 11.6 Fireside Chat

In this chapter we developed a custom Terraform Petstore provider. This provider relied on a remote API and a client SDK library written in golang. Instead of directly calling the API to provision resources, customers can now use Terraform to manage resources on their behalf.

Where I see developing custom providers fitting best is with micro-APIs and self-service platforms. If you are a service owner, you probably already make your service available to customers through a RESTful API (Note that Serverless patterns are a great fit here because they make it cheap and easy to deploy single purpose APIs). Unfortunately, most customers do not want to go through the trouble of learning how to authenticate against your API to provision resources. This can lower the adoption rate of even a great self-service platform. By writing a Terraform provider for your API you make it easy for people to start using your API with little to no knowledge of the API or underlying protocols and procedures.

Writing your own provider is certainly a large investment, but it's also not as challenging as some people make it out to be. By now you should have the skills necessary to comfortably approach writing your own provider and/or make contributions to existing providers.

## 11.7 Summary

- Terraform providers make it easy for people to use APIs without knowing how they work. In this spirit, you should always design providers to be as user friendly as possible.
- Providers expose resources and data sources to Terraform. These are implemented as functions, referenced by the provider schema.
- Managed resources implement `CRUD` operations - `CREATE`, `READ`, `UPDATE` and `DELETE`. These methods are invoked when the relevant lifecycle event is triggered.
- Acceptance testing means writing tests for the provider schema and any resources exposed by the provider. Acceptance testing hardens code and is crucial for production readiness.
- Providers are built like any other golang program. You should setup a CI/CD pipeline to automate building, testing, publishing, and distributing the provider.

# 12

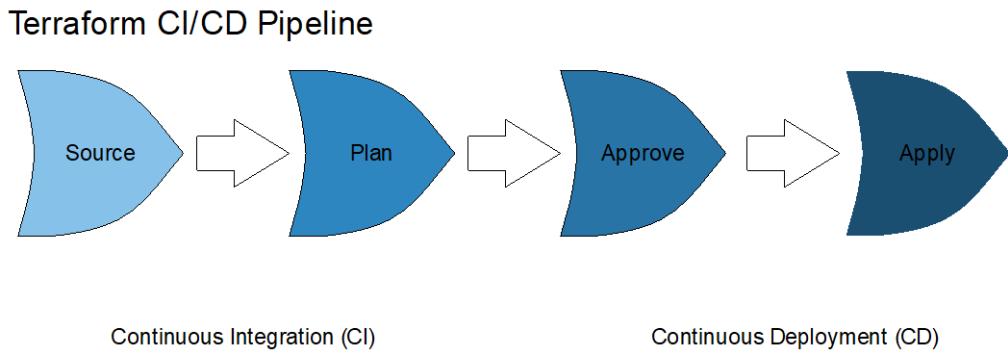
## *Terraform in Automation*

### This chapter covers:

- Designing workflows for running Terraform at scale
- Developing a CI/CD pipeline for automating Terraform deployments
- Generating Terraform configuration code
- Toggling dynamic blocks with a conditional expression

If you are struggling with how to run Terraform at scale, rest easy friend, because this chapter is for you. Until now, I have assumed you are deploying Terraform from your local machine. This is a reasonable assumption for individuals and even small teams (as long as you are using a remote state backend). Large teams and organizations with many individual contributors, on the other hand, would benefit from running Terraform in automation.

As you may recall from chapter 6, HashiCorp already has two products designed for running Terraform in automation: *Terraform Cloud* and *Terraform Enterprise*. They are basically the same product; Terraform Cloud is just the managed SaaS offering of Terraform Enterprise. In this chapter we will develop a CI/CD pipeline for deploying Terraform workspaces in automation, modeled after the design of Terraform Enterprise. The stages of the CI/CD pipeline are shown in figure 12.1.



**Figure 12.1** A CI/CD pipeline for Terraform deployments with four stages. Changes to configuration code stored in a VCS source repository triggers a “terraform plan” to be run. If the plan succeeds, a manual approval is required before the changes are applied into production.

By the end of this chapter you will have the skills necessary to automate Terraform deployments using a CI/CD pipeline. I will also give some advice on how to structure more complex Terraform CI/CD pipelines, although the actual implementation is outside the scope of what is covered here.

## 12.1 Poor Man’s Terraform Enterprise

Why develop a custom solution for running Terraform in automation when HashiCorp already has Terraform Enterprise? Two reasons: ownership and cost.

- **Ownership** – by owning the code, you are able to design the solution the works for you and troubleshoot when something goes wrong.
- **Cost** – Terraform Enterprise is not free. You can save a lot of money by forgoing the licensing fees and developing a homegrown solution (~ several hundred thousand dollars)

We will start by reverse engineering Terraform Enterprise, and from there come up with a plan of attack.

### 12.1.1 Reverse Engineering Terraform Enterprise

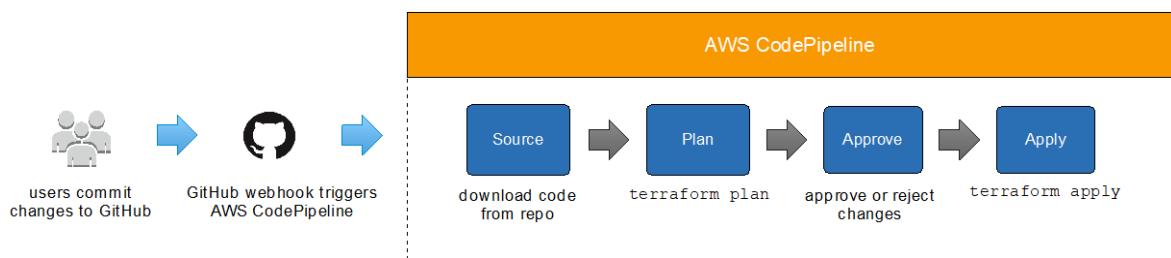
All the features of Terraform Enterprise fall under one of two main categories: collaboration and automation. Collaboration features are those designed to help people share and develop Terraform with each other, while automation features are those which make it easier to integrate Terraform as part of a CI/CD pipeline.

Theme	Key Features
Collaboration	<ul style="list-style-type: none"> <li>State Management (storage, viewing, history, and locking)</li> <li>Web UI for viewing and approving runs</li> <li>Collaborative Runs</li> <li>Private Module Registry</li> <li>Sentinel and “Policy as Code”</li> </ul>
Automation	<ul style="list-style-type: none"> <li>Version Control System (VCS) Integration</li> <li>GitOps workflow</li> <li>Remote CLI Operations</li> <li>Notifications for run events</li> <li>Full HTTP API for integration with other tools and services</li> </ul>

**Table 12.1 Key features of Terraform Enterprise, categorized by theme**

Our poor man’s Terraform Enterprise will support all of the collaboration and automation features of Terraform Enterprise as listed in Table 12.1, with the exception of remote operations and Sentinel “Policy as Code”. This is because remote operations are not supported by open source Terraform, while Sentinel is not part of the open source community at all. We will talk more about Sentinel in chapter 12, because it’s still relevant as it relates to secrets management.

A conceptual diagram of what we are going to build is shown in figure 12.2. It’s a concrete implementation of the more general Terraform CI/CD workflow depicted earlier. The basic idea is that users will check in configuration code to a GitHub repository, which then fires a webhook that triggers an execution of AWS CodePipeline to perform Terraform actions like plan and apply.



**Figure 12.2 A concrete implementation of the more general Terraform CI/CD workflow. Users check in configuration changes to a GitHub source repository, which in turn triggers an execution of AWS CodePipeline. The pipeline has four stages: source, plan, approve and apply.**

AWS CodePipeline is a GitOps service similar to GCP Cloud Build and Azure DevOps. It supports having multiple stages. These stages can either execute predefined tasks or run

custom code as defined by a YAML build specification file. We will have four stages in our CI/CD pipeline, one for creating a webhook/ downloading source code from a GitHub repository (Source), one for running `terraform plan` (Plan), a third for obtaining manual approval (Approve), and the last for running `terraform apply` (Apply). Having a manual approval stage is important to allow stakeholders (i.e. anyone that has an invested interest in the outcome of a Terraform deployment) read the output of `terraform plan` before approving an apply.

Figure 12.3 is a closeup of what happens inside the pipeline.

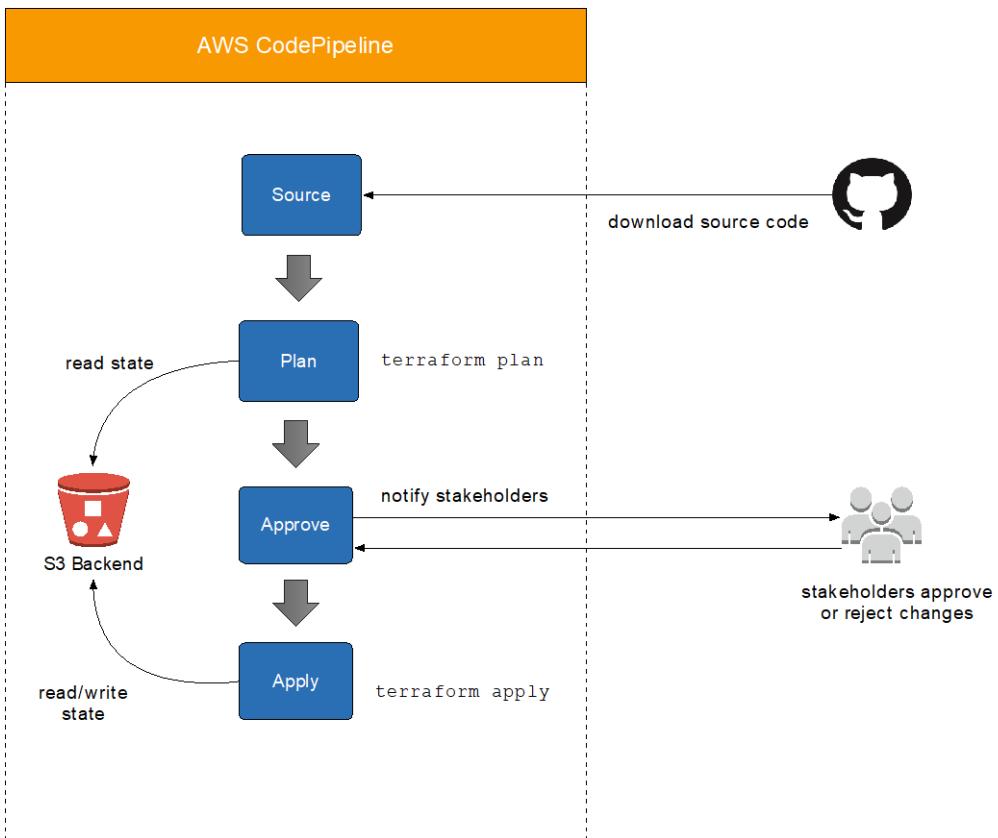
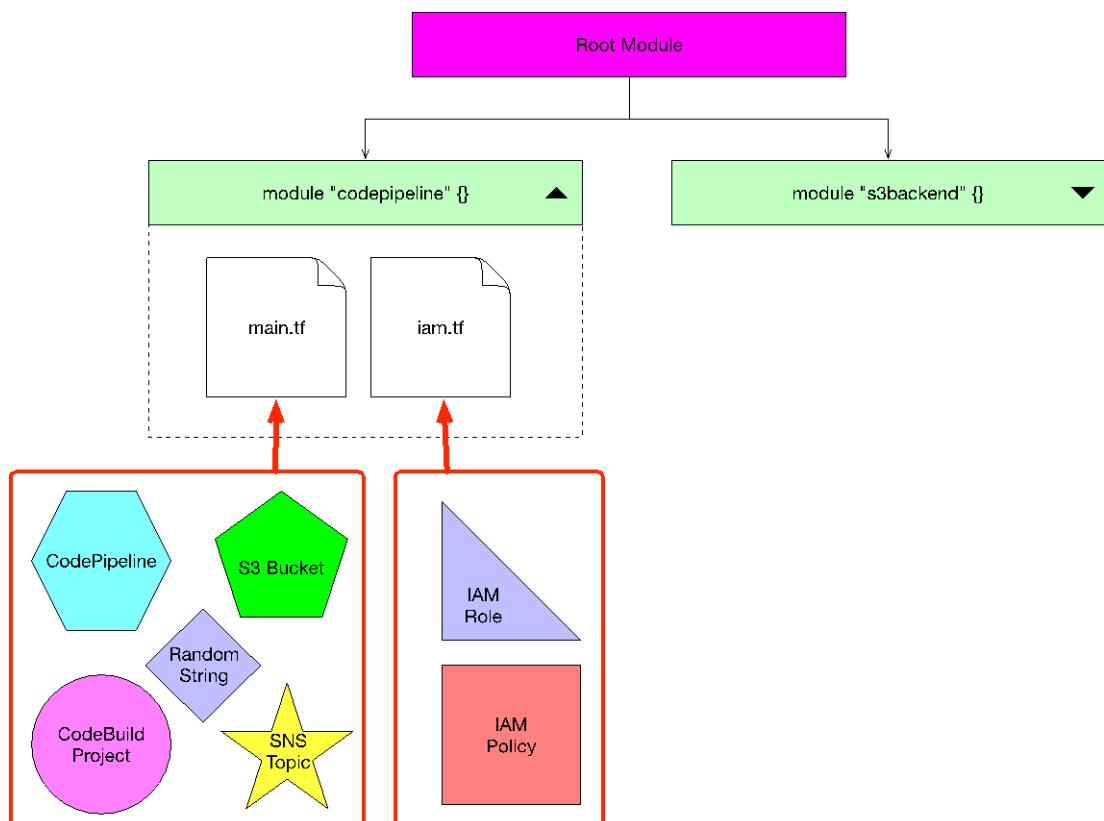


Figure 12.3 Closeup of the pipeline. “Source” downloads source code from GitHub. “Plan” runs a `terraform plan`. “Approve” notifies stakeholders (people with an invested interest in the outcome of the deployment) to manually approve or reject changes. “Apply” runs `terraform apply`.

### 12.1.2 Detailed Engineering

Our goal is to design a Terraform project that can manage CI/CD pipeline(s) as code for automating other kinds of Terraform deployments. Another way of thinking about this is that we are using Terraform to manage Terraform. In this section we will walk through the overall structure and design of the project, and in the next few sections we are going to build and deploy it.

At the root level we will declare two modules, one for deploying AWS CodePipeline and another for deploying an S3 remote backend. The `codepipeline` module is going to contain all the resources and auxiliary resources for provisioning the pipeline. These include: IAM resources, CodeBuild projects, an SNS topic and an S3 bucket. The `s3backend` module deploys a remote state backend for securely storing, encrypting, and locking Terraform state files. We will not go into any detail of what goes into the `s3backend` module, as this was covered in chapter 6. Figure 12.4 depicts the project's overall structure.



**Figure 12.4** At the root level are two modules: “codepipeline” and “s3backend”. The former defines the resources for creating a CI/CD pipeline in AWS CodePipeline while the latter provisions an S3 remote backend

(see chapter 6 for more details on the `s3backend` module)

**NOTE** this project combines a nested module structure with a flat module structure. Normally I would recommend sticking to one or the other, but it is not wrong to incorporate both, so long as the code is clear and understandable.

Our completed project directory structure will contain eight files spread over three directories:

```
$ tree -C
.
├── main.tf
└── modules
    └── codebuild
        ├── iam.tf
        ├── main.tf
        └── templates
            ├── backend.json
            ├── buildspec_apply.yml
            └── buildspec_plan.yml
        └── variables.tf
└── terraform.tfvars

3 directories, 8 files
```

## 12.2 Beginning at the Root

To begin development of our CI/CD pipeline, we need to create a new Terraform workspace. In this workspace we will declare the `s3backend` and `codepipeline` modules, following a nested module structure. Note that the `s3backend` module is an existing module, created in chapter 6 and published to the public module registry, so requires no implementation. Conversely, the `codepipeline` module, does require implementation, and we will go through it all in the next section.

### 12.2.1 Writing the Module Wrapper Code

Following a nested module structure means breaking logical components into separate modules. At the top level will be a root module that exclusively contains references to other modules (which may be sourced locally, or remotely). The code that goes in the root module is sometimes referred to as “module wrapper code”, because it is simply boilerplate code that links everything together.

Let’s start from the root module. Create a new Terraform workspace. Do this by making a new directory with a `main.tf` file. `main.tf` will contain the wrapper code for deploying the two modules.

Add the code from Listing 12.1 into `main.tf`.

**Listing 12.1 main.tf**

```

variable "aws" {
  type = object({access_key = string, secret_key = string, region = string})
}

variable "vcs_repo" {
  type = object({identifier = string, branch = string, oauth_token = string})
}

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

module "s3backend" {
  source = "scottwinkler/s3backend/aws" #A
}

module "codepipeline" { #B
  source          = "./modules/codepipeline"
  name            = "terraform-in-action"
  vcs_repo        = var.vcs_repo
  environment = {
    AWS_ACCESS_KEY_ID      = var.aws.access_key
    AWS_SECRET_ACCESS_KEY = var.aws.secret_key
  }
  s3_backend_config = module.s3backend.config
}

```

#A Module "s3backend" requires no additional configuration because the defaults are fine as is.  
#B we will go through what the input variables are in the next section

**NOTE** we will skip creating the `terraform.tfvars` file for now, as it requires some setup work that isn't important just yet. We will return to it when we are ready to deploy.

## 12.3 Developing a Terraform CI/CD Pipeline

Here is where we define the module that provisions the CI/CD pipeline.

### 12.3.1 Configuring Input Variables

Since this will be a locally sourced module, create a `./modules/codepipeline` directory for storing the module code and switch into it. In this directory create a `variables.tf` file and add in the code from Listing 12.2. This file is responsible for declaring all the input variables that `codepipeline` uses.

**Listing 12.2 variables.tf**

```

variable "name" {
  type    = string
  default = "terraform"
  description = "A project name to use for resource mapping"
}

```

```

variable "auto_apply" {
  type    = bool
  default = false
  description = "Whether to automatically apply changes when a Terraform plan is successful.
                  Defaults to false."
}

variable "terraform_version" {
  type    = string
  default = "latest"
  description = "The version of Terraform to use for this workspace. Defaults to the latest
                  available version."
}

variable "working_directory" {
  type    = string
  default = "."
  description = "A relative path that Terraform will execute within. Defaults to the root of
                  your repository."
}

variable "vcs_repo" {
  type = object({ identifier = string, branch = string, oauth_token = string })
  description = "Settings for the workspace's VCS repository."
}

variable "environment" {
  type    = map(string)
  default = {}
  description = "A map of environment variables to use for this workspace"
}

variable "s3_backend_config" {
  type = object({
    bucket      = string,
    region      = string,
    role_arn    = string,
    dynamodb_table = string,
  })
  description = "Settings for configuring the S3 remote backend"
}

```

### 12.3.2 IAM Roles and Policies

AWS CodeBuild (a service used by the “Plan” and “Apply” stages) and AWS CodePipeline (a service used to construct the pipeline) each require their own IAM role and policy. CodeBuild’s IAM role allows it to create logs and read from a certain S3 bucket, while CodePipeline’s role allows it to perform basic execution tasks. The details are not important or particularly interesting. We will break it out into its own file so it will not detract from the rest of the scenario (following a flat module structure). Create an `iam.tf` file and add in the code from Listing 12.3.

#### **Listing 12.3 iam.tf**

```
resource "aws_iam_role" "codebuild_role" {
```

```

name          = "${local.namespace}-codebuild"
assume_role_policy = <<-EOF
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "codebuild.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
EOF
}

resource "aws_iam_role_policy" "codebuild_policy" {
  role    = aws_iam_role.codebuild_role.name
  policy = <<-EOF
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Resource": [
        "*"
      ],
      "Action": [
        "logs>CreateLogGroup",
        "logs>CreateLogStream",
        "logs:PutLogEvents"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "s3:GetObject",
        "s3:GetObjectVersion",
        "s3:GetBucketVersioning",
        "s3:PutObject"
      ],
      "Resource": [
        "${aws_s3_bucket.codepipeline_bucket.arn}",
        "${aws_s3_bucket.codepipeline_bucket.arn}/*"
      ]
    }
  ]
}
EOF
}

resource "aws_iam_role" "codepipeline_role" {
  name          = "${local.namespace}-pipeline-role"
  assume_role_policy = <<-EOF
{
  "Version": "2012-10-17",
  "Statement": [
    {

```

```

    "Effect": "Allow",
    "Principal": {
        "Service": "codepipeline.amazonaws.com"
    },
    "Action": "sts:AssumeRole"
}
]

}

EOF
}

resource "aws_iam_role_policy" "codepipeline_policy" {
    name = "codepipeline_policy"
    role = aws_iam_role.codepipeline_role.id

    policy = <<-EOF
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "s3:GetObject",
                "s3:GetObjectVersion",
                "s3:GetBucketVersioning",
                "s3:PutObject"
            ],
            "Resource": [
                "${aws_s3_bucket.codepipeline_bucket.arn}",
                "${aws_s3_bucket.codepipeline_bucket.arn}/*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "sns:Publish"
            ],
            "Resource": "${aws sns topic.codepipeline.arn}"
        },
        {
            "Effect": "Allow",
            "Action": [
                "codebuild:BatchGetBuilds",
                "codebuild:StartBuild"
            ],
            "Resource": "*"
        }
    ]
}
EOF
}

```

### 12.3.3 Building the Plan and Apply Stages

In this section we will build the “Plan” and “Apply” stages of the pipeline. Both of these stages utilize AWS CodeBuild, which is why we are starting with them as a pair. Before we begin, let’s

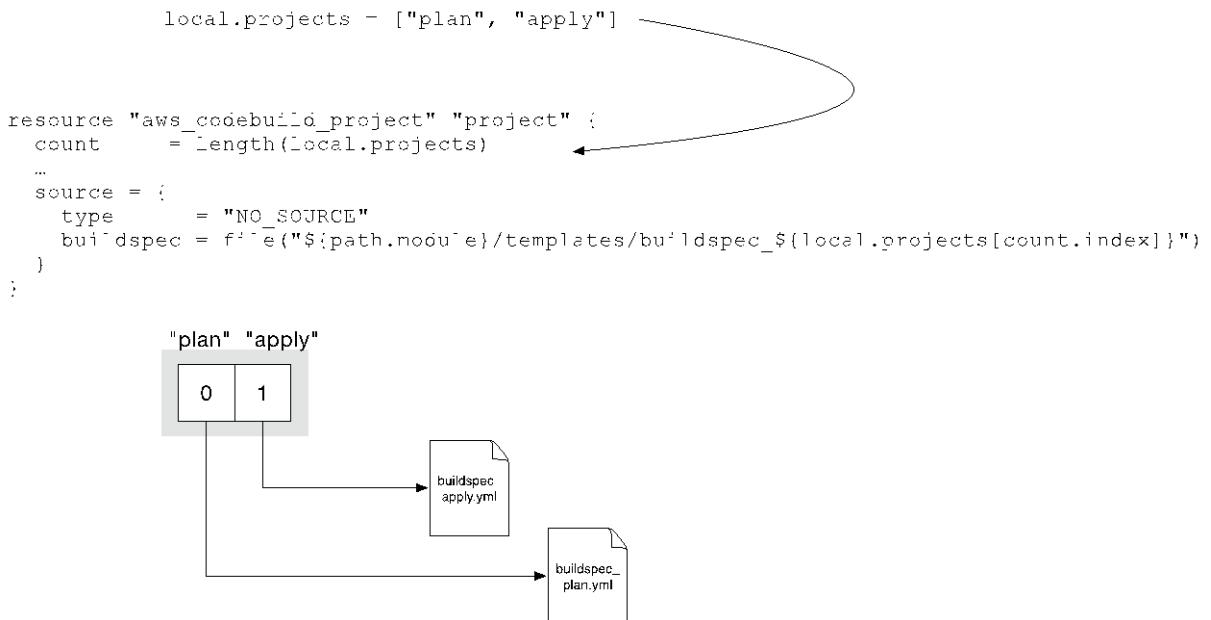
add a `random_string` resource to prevent namespace collisions (like we did in Chapter 5). Add in the code from Listing 12.4 to `main.tf`.

#### **Listing 12.4 main.tf**

```
resource "random_string" "rand" {
  length  = 24
  special = false
  upper   = false
}

locals {
  namespace = substr(join("-", [var.name, random_string.rand.result]), 0, 24)
}
```

Let's now configure a AWS CodeBuild project for the "Plan" and "Apply" stages of the pipeline ("Source" and "Approve" do not use AWS CodeBuild). As the CodeBuild projects for "Plan" and "Apply" are nearly identical, we'll template it out to make the code more concise and extendable (see figure 12.5).



**Figure 12.5 AWS CodePipeline is made up of multiple stages, two of which depend on AWS CodeBuild projects ("Plan" and "Apply). Since these two stages are nearly identical, we will leverage template files to make our lives easier**

Add the code from Listing 12.5 into `main.tf` to provision the two AWS CodeBuild project resources:

**Listing 12.5 main.tf**

```

locals {
  projects = ["plan", "apply"]
}

resource "aws_codebuild_project" "project" {
  count      = length(local.projects)
  name       = "${local.namespace}-${local.projects[count.index]}"
  service_role = aws_iam_role.codebuild_role.id

  artifacts {
    type = "NO_ARTIFACTS"
  }

  environment {
    compute_type = "BUILD_GENERAL1_SMALL"
    image        = "hashicorp/terraform:${var.terraform_version}" #A
    type         = "LINUX_CONTAINER"
  }

  source {
    type      = "NO_SOURCE"
    buildspec = file("${path.module}/templates/buildspec_${local.projects[count.index]}.yml")
  }
}

```

#A this points to an image published by HashiCorp

Notice, in Listing 12.5, that `var.terraform_version` selects the tagged release of a `hashicorp/terraform` image to use for the container runtime. HashiCorp maintains this image and has a tagged released for each version of Terraform, with the “latest” version pointing to the current release. It’s a lightweight Alpine Linux image that has the Terraform binary baked in. The reason why we are using it is to save us from having to download the Terraform binary at runtime (a potentially slow operation).

A build specification (`buildspec`) files contains the collection of build commands and related settings that AWS CodeBuild will execute on. Create a `./templates` folder to put the `buildspec` files for “Plan” and “Apply” into.

Create a `buildspec_plan.yml` that will be used by the “Plan” stage. In this file add the code from Listing 12.6.

**Listing 12.6 buildspec\_plan.yml**

```

version: 0.2
phases:
  build:
    commands:
      - cd $WORKING_DIRECTORY
      - echo $BACKEND >> backend.tf.json
      - terraform init
      - |
        if [[ "$CONFIRM_DESTROY" == "0" ]]; then #A
          terraform plan
        else

```

```
    terraform plan -destroy
fi
```

#A if CONFIRM\_DESTROY is zero then run a terraform plan, otherwise run a destroy plan

As you can see, the “Plan” stage does a bit more than simply run `terraform plan`. Here is the logic, in case you are wondering:

1. Switch into the working directory of the source code as specified by the `WORKING_DIRECTORY` environment variable. This defaults to the current working directory.
2. Write a `backend.tf.json` file. This file is what configure the S3 backend for remote state storage
3. Initialize Terraform with a `terraform init`
4. Perform a `terraform plan` if `CONFIRM_DESTROY` is set to zero, otherwise perform a destroy plan (`terraform plan -destroy`)

**TIP** if you had a use case to install custom Terraform providers that are not available in the provider registry, you could do so by downloading them anytime before `terraform init`

The build specification for the Apply stage is similar to the Plan and in shown in Listing 12.7. Create a `buildspec_apply.yml` in `./templates` folder with the code from Listing 12.7.

**NOTE** It is possible to create a general `buildspec` that could work for both “Plan” and “Apply”, however, I think it makes the code harder to understand

#### **Listing 12.7 buildspec\_apply.yml**

```
version: 0.2
phases:
  build:
    commands:
      - cd $WORKING_DIRECTORY
      - echo $BACKEND >> backend.tf.json
      - terraform init
      - |
        if [[ "$CONFIRM_DESTROY" == "0" ]]; then
          terraform apply -auto-approve
        else
          terraform destroy -auto-approve
        fi
```

#### **12.3.4 Configuring Environment Variables**

Users are able to set environment variables on the container runtime by passing a map of strings into the `environment` input argument. The reason a user might choose to set environment variables, rather than solely rely on a variables definition file, is typically for

setting secrets that should not be checked into version controlled source repositories, like AWS access and secret access keys.

**TIP** environment variables are not the only way to get secrets into Terraform. We will discuss some of the other options in chapter 12

Environment variables passed by users are merged with default environment variables into a new map. This new map will then be transformed into a JSON list of objects with a for expression. The output of the for expression is used to configure environment variables on the “Plan” and “Apply” stages. Figure 12.6 is a visual depiction of how user supplied environment variables are combined with default environment variables and then transformed into a list of JSON objects (as per AWS CodePipeline’s documentation<sup>1</sup>).

**NOTE** There are two additional ways to configure environment variables besides setting them in AWS CodePipeline: 1) via the buildspec file, and 2) as an attribute of aws\_codebuild\_project

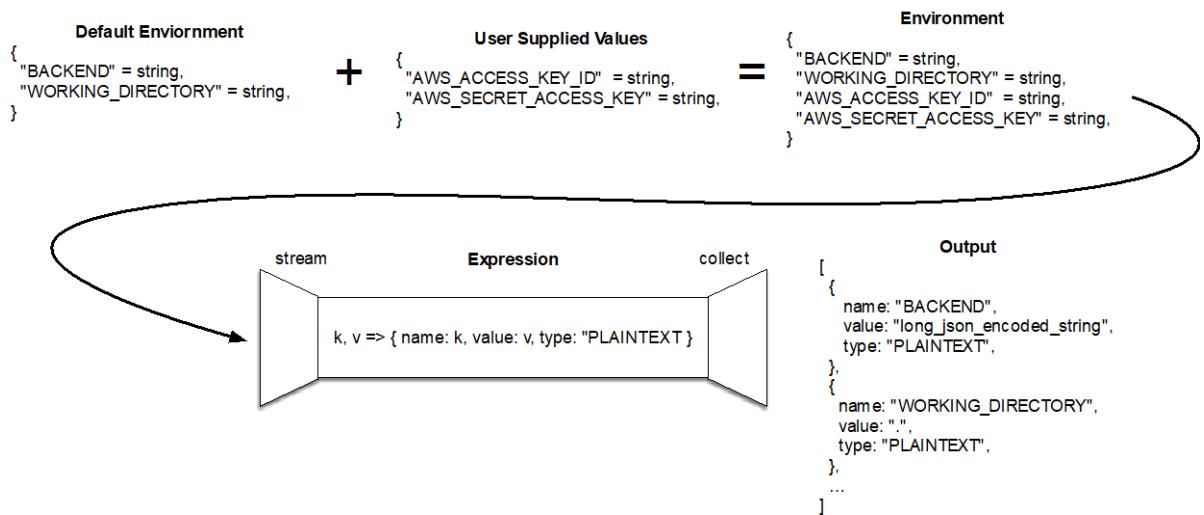


Figure 12.6 User supplied environment variables are merged with default environment variables into a new map. The map is then converted with a for expression into a JSON list of objects and used to configure AWS CodePipeline

<sup>1</sup> <https://docs.aws.amazon.com/codepipeline/latest/userguide/action-reference-CodeBuild.html>

The code that declares `default_environment` as a local value, merges it with `var.environment`, and transforms it into a JSON list of objects is shown in Listing 12.8.

#### **Listing 12.8 main.tf**

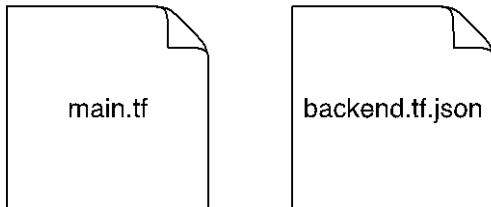
```
locals {
  backend = templatefile("${path.module}/templates/backend.json", { config :
    var.s3_backend_config, name : local.namespace })

  default_environment = {
    TF_IN_AUTOMATION = "1"
    TF_INPUT          = "1"
    CONFIRM_DESTROY   = "0"
    WORKING_DIRECTORY = var.working_directory
    BACKEND           = local.backend,
  }

  environment = jsonencode([for k, v in merge(local.default_environment, var.environment) : {
    name : k, value : v, type : "PLAINTEXT" }])
}
```

As you can see, there are five default environment variables: `TF_IN_AUTOMATION`, `TF_INPUT`, `CONFIRM_DESTROY`, `WORKING_DIRECTORY` and `BACKEND`. The first two are for configuring the Terraform runtime, the third is a flag for triggering a destroy run, the fourth sets the current working directory, and the fifth configures the remote state backend. We will generate code for configuring the remote state backend with a template file and then write the output to `backend.tf.json` prior to initializing Terraform. This is helpful because now users do not need to check in backend configuration code to version control.

```
echo $BACKEND >> backend.tf.json
```



**Figure 12.7** Before Terraform is initialized, a `backend.tf.json` file is created to configure the remote state backend. This makes it so users do not have to check in a `backend.tf` file to GitHub.

The template file that generates backend configuration code using a Terraform settings block is shown in Listing 12.9. Add this to a new `backend.json` file under the `./templates` directory.

**Listing 12.9 backend.json**

```
{
  "terraform": {
    "backend": {
      "s3": {
        "bucket": "${config.bucket}",
        "key": "aws/${name}",
        "region": "${config.region}",
        "encrypt": true,
        "role_arn": "${config.role_arn}",
        "dynamodb_table": "${config.dynamodb_table}"
      }
    }
  }
}
```

**Why JSON and not HCL?**

Most Terraform configurations are written in HCL because it's an easy language for humans to read and understand, but it's also possible to write Terraform configuration in JSON instead. This alternative syntax (suffixed with the `.tf.json` extension) is normally reserved for automation purposes, because it is more machine friendly than HCL, and because many programming languages already have native libraries for processing JSON. As I pointed out in chapter 5, programmatically generated code is generally discouraged, as modules are the preferred mechanism for code reuse, but this case is an exception.

**12.3.5 Declaring the Pipeline as Code**

There are a few other miscellany resources that CodePipeline relies on. First, there is an S3 bucket that is used to cache artifacts between build stages (just part of how CodePipeline works). Second, there's an SNS topic that will be used by the "Approve" stage to send notifications when a manual approval is required. Currently these notifications go nowhere, but SNS can be configured to send notifications anywhere you like.

**TIP** SNS can be configured to send emails to a mailing list (via SES), texts to a cellphone (via SMS), or notifications to a Slack channel (via Chimebot). Unfortunately, all of these require some degree of manual setup, which is why I have omitted them from this scenario

Add the code from Listing 12.10 into `main.tf` to declare an S3 bucket and SNS topic.

**Listing 12.10 main.tf**

```
resource "aws_s3_bucket" "codepipeline_bucket" {
  bucket      = "${local.namespace}-codepipeline-bucket"
  acl         = "private"
  force_destroy = true
}

resource "aws sns topic" "codepipeline" {
  name = "${local.namespace}-pipeline-topic"
```

```
}
```

Finally, we are ready to declare the pipeline! As a reminder, there are four stages that makeup the pipeline. Here are the four stages and what they do:

5. **Source** – Creates a webhook and downloads source code from a GitHub repository
6. **Plan** – Runs a terraform plan with the source code
7. **Approve** – Waits for manual approval
8. **Apply** – Runs a terraform apply with the source code

Add the code from Listing 12.11 for declaring the pipeline into `main.tf`.

### **Listing 12.11 main.tf**

```
resource "aws_codepipeline" "codepipeline" {
  name      = "${local.namespace}-pipeline"
  role_arn = aws_iam_role.codepipeline_role.arn

  artifact_store {
    location = aws_s3_bucket.codepipeline_bucket.bucket
    type     = "S3"
  }

  stage {
    name = "Source"

    action {
      name          = "Source"
      category     = "Source"
      owner         = "ThirdParty"
      provider      = "GitHub"
      version       = "1"
      output_artifacts = ["source_output"]

      configuration = {
        Owner      = split("/", var.vcs_repo.identifier)[0]
        Repo       = split("/", var.vcs_repo.identifier)[1]
        Branch     = var.vcs_repo.branch
        OAuthToken = var.vcs_repo.oauth_token
      }
    }
  }

  stage {
    name = "Plan"

    action {
      name          = "Plan"
      category     = "Build"
      owner         = "AWS"
      provider      = "CodeBuild"
      input_artifacts = ["source_output"]
      version       = "1"

      configuration = {
        ProjectName      = aws_codebuild_project.project[0].name
      }
    }
  }
}
```

```

        EnvironmentVariables = local.environment
    }
}
}

dynamic "stage" {
    for_each = ! var.auto_apply ? [1] : []
    content {
        name = "Approval"

        action {
            name      = "Approval"
            category = "Approval"
            owner     = "AWS"
            provider  = "Manual"
            version   = "1"

            configuration = {
                CustomData      = "Please review output of plan and approve"
                NotificationArn = aws_sns_topic.codepipeline.arn
            }
        }
    }
}

stage {
    name = "Apply"

    action {
        name      = "Apply"
        category = "Build"
        owner     = "AWS"
        provider  = "CodeBuild"
        input_artifacts = ["source_output"]
        version   = "1"

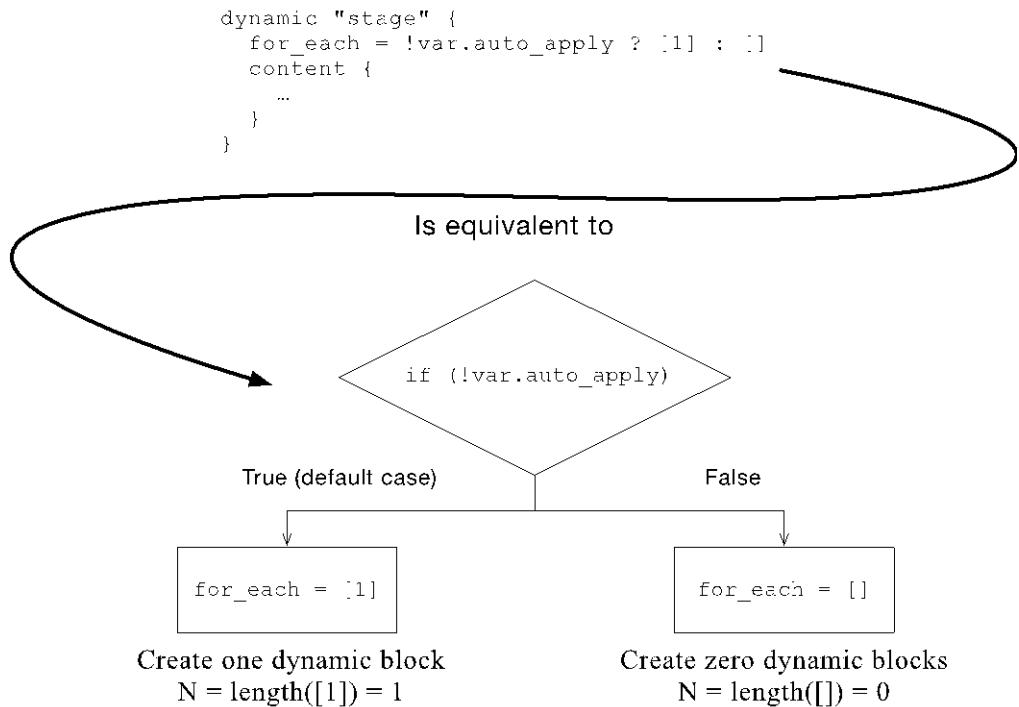
        configuration = {
            ProjectName      = aws_codebuild_project.project[1].name
            EnvironmentVariables = local.environment
        }
    }
}
}

```

An interesting thing to point out in the pipeline code is the use of a dynamic block with a feature flag to toggle provisioning the “Approve” stage. If the feature flag `var.auto_apply` is set to `true`, the stage won’t be created, because `for_each` is set to iterate over an empty list (similar to setting `count=0` on a resource). Conversely, if left at the default values, it will be created (because `for_each` iterates over a list of length one). This allows users to optionally skip “Approve” and go directly from “Plan” to “Apply”.

**WARNING!** it is not recommended to turn off manual approval for anything mission critical! It’s like performing a `terraform apply -auto-approve` without even checking the results of the plan first. There should always be at least one human that is verifying the results of the plan before changes are applied

The logic for toggling a dynamic block with a feature flag is shown in figure 12.8



**Figure 12.8** If `var.auto_apply` is set to true then `!var.auto_apply` evaluates to false, so `for_each` will iterate over an empty list, meaning no blocks will be created. If left at the default value of false, `!var.auto_apply` evaluates to true, so `for_each` will iterate over a list of length one, meaning exactly one block will be created.

### 12.3.6 Touching Base

For your reference, the complete code for `main.tf` is shown in Listing 12.12.

#### Listing 12.12 complete main.tf

```

resource "random_string" "rand" {
  length  = 24
  special = false
  upper   = false
}

locals {
  namespace = substr(join("-", [var.name, random_string.rand.result]), 0, 24)
  projects  = ["plan", "apply"]
}

resource "aws_codebuild_project" "project" {

```

```

count      = length(local.projects)
name       = "${local.namespace}-${local.projects[count.index]}"
service_role = aws_iam_role.codebuild_role.id

artifacts {
    type = "NO_ARTIFACTS"
}

environment {
    compute_type = "BUILD_GENERAL1_SMALL"
    image        = "hashicorp/terraform:${var.terraform_version}"
    type         = "LINUX_CONTAINER"
}

source {
    type      = "NO_SOURCE"
    buildspec = file("${path.module}/templates/buildspec_${local.projects[count.index]}.yml")
}
}

locals {
    backend = templatefile("${path.module}/templates/backend.json", { config :
        var.s3_backend_config, name : local.namespace })
    default_environment = {
        TF_IN_AUTOMATION = "1"
        TF_INPUT          = "1"
        WORKING_DIRECTORY = var.working_directory
        BACKEND           = local.backend,
    }
    environment = jsonencode([for k, v in merge(local.default_environment, var.environment) : {
        name : k, value : v, type : "PLAINTEXT" }])
}
}

resource "aws_s3_bucket" "codepipeline_bucket" {
    bucket      = "${local.namespace}-codepipeline-bucket"
    acl         = "private"
    force_destroy = true
}

resource "aws sns topic" "codepipeline" {
    name = "${local.namespace}-pipeline-topic"
}

resource "aws_codepipeline" "codepipeline" {
    name      = "${local.namespace}-pipeline"
    role_arn = aws_iam_role.codepipeline_role.arn

    artifact_store {
        location = aws_s3_bucket.codepipeline_bucket.bucket
        type     = "S3"
    }

    stage {
        name = "Source"

        action {
            name      = "Source"
            category = "Source"
            owner    = "ThirdParty"
        }
    }
}

```

```

provider      = "GitHub"
version       = "1"
output_artifacts = ["source_output"]

configuration = {
  Owner        = split("/", var.vcs_repo.identifier)[0]
  Repo         = split("/", var.vcs_repo.identifier)[1]
  Branch       = var.vcs_repo.branch
  OAuthToken   = var.vcs_repo.oauth_token
}
}

stage {
  name = "Plan"

  action {
    name          = "Plan"
    category     = "Build"
    owner         = "AWS"
    provider      = "CodeBuild"
    input_artifacts = ["source_output"]
    version       = "1"

    configuration = {
      ProjectName      = aws_codebuild_project.project[0].name
      EnvironmentVariables = local.environment
    }
  }
}

dynamic "stage" {
  for_each = ! var.auto_apply ? [1] : []
  content {
    name = "Approval"

    action {
      name          = "Approval"
      category     = "Approval"
      owner         = "AWS"
      provider      = "Manual"
      version       = "1"

      configuration = {
        CustomData      = "Please review output of plan and approve"
        NotificationArn = aws sns topic.codepipeline.arn
      }
    }
  }
}

stage {
  name = "Apply"

  action {
    name          = "Apply"
    category     = "Build"
    owner         = "AWS"
    provider      = "CodeBuild"
  }
}

```

```
    input_artifacts = ["source_output"]
    version         = "1"

    configuration = {
        ProjectName      = aws_codebuild_project.project[1].name
        EnvironmentVariables = local.environment
    }
}
}
```

### 12.3.7 Deploying to AWS

We still need to set variables for the project before we are ready to deploy. Switch back into the root directory of the project and create a `terraform.tfvars` file. Listing 12.13 shows what this file will look like, although you will need to replace the noted values with your own.

### **Listing 12.13** terraform.tfvars

```
aws = {
    access_key = "AKIAITESI2XGPI5F" #A
    secret_key = "x2TZm1kBkfH4Z6P" #A
    region     = "us-west-2"
}

vcs_repo = {
    branch      = "master" #B
    identifier  = "swinkler/test_deploy" #B
    oauth_token = "b4c5e88c29a192cd69d150" #C
}
```

#A your AWS access and secret access keys go here

#B the branch and identifier of the GitHub source repository

#C personal access token for managing GitHub webhooks

**NOTE** `vcs_repo` configures the connection to a GitHub repository: `identifier` is the repository name, `branch` is the branch name, and `oauth_token` is a personal access token used to manage webhooks.

**NOTE** AWS\_ACCESS\_KEY\_ID and AWS\_SECRET\_ACCESS\_KEY configure AWS for Terraform running in the pipeline. They do not have to be the same as the credentials used to configure the AWS provider, in fact, they are usually different.

## Obtaining a GitHub Personal Access Token (OAuth)

You will need to obtain a personal access token for connecting the “Source” stage to GitHub. The token requires the following permissions:

- **repo**: Grants full control of private repositories.
  - **repo:status**: Grants access to commit statuses.
  - **admin:repo\_hook**: Grants full control of repository hooks.

Personal access tokens can be created through GitHub's UI by navigating to >Settings>Developer Settings>Personal Access Tokens. Consult the official GitHub documentation<sup>2</sup> for more information on generating personal access tokens.

Once you have set the variables, run a `terraform apply`.

```
$ terraform apply
...
# module.s3backend.random_string.rand will be created
+ resource "random_string" "rand" {
  + id      = (known after apply)
  + length  = 24
  + lower   = true
  + min_lower = 0
  + min_numeric = 0
  + min_special = 0
  + min_upper = 0
  + number   = true
  + result   = (known after apply)
  + special   = false
  + upper    = false
}

Plan: 18 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?
Terraform will perform the actions described above.
Only 'yes' will be accepted to approve.

Enter a value:
```

After confirming the apply, it should only take a minute or two for the pipeline to be deployed.

```
module.codepipeline.aws_codebuild_project.project[0]: Creation complete after 18s
  [id=arn:aws:codebuild:us-west-2:215974853022:project/terraform-in-action-iuozi]
module.codepipeline.aws_codepipeline.codepipeline: Creating...
module.codepipeline.aws_codepipeline.codepipeline: Creation complete after 1s [id=terraform-
in-action-iuozi-pipeline]

Apply complete! Resources: 18 added, 0 changed, 0 destroyed.
```

Figure 12.9 depicts the deployed pipeline as viewed from the AWS console.

**NOTE** the pipeline has failed on the first stage because the source repository doesn't exist yet. We will create it in the next section

---

<sup>2</sup> <https://help.github.com/en/github/authenticating-to-github/creating-a-personal-access-token-for-the-command-line>

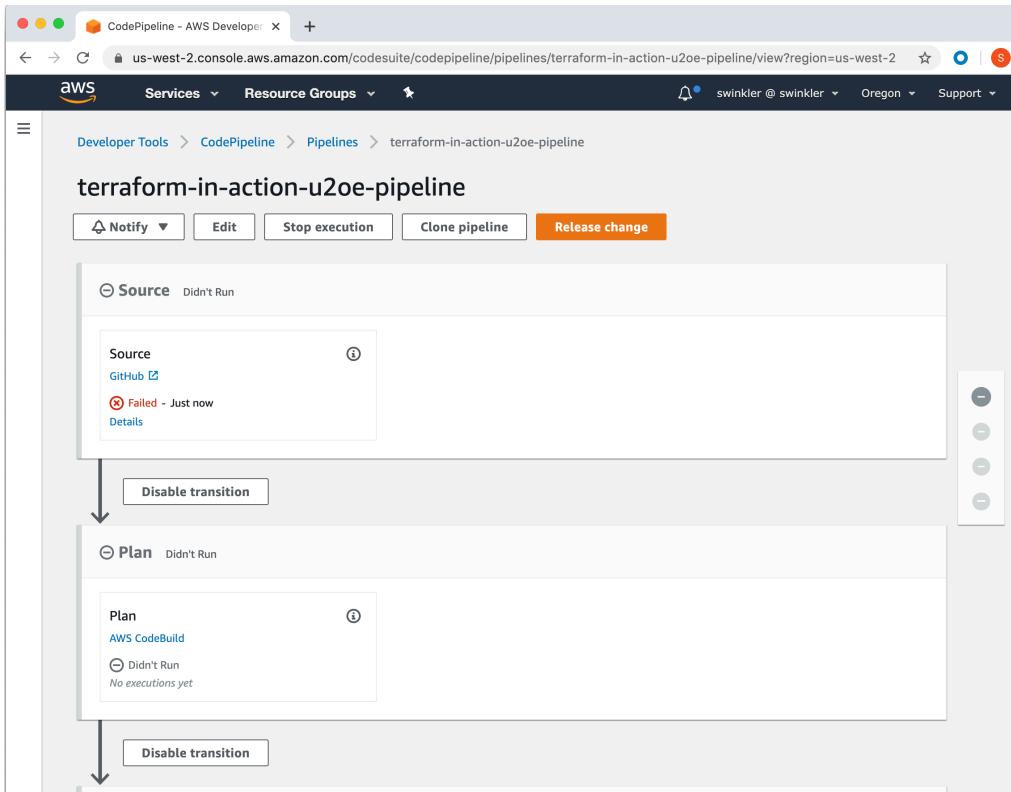


Figure 12.9 The deployed AWS CodePipeline, as viewed from the AWS console. Currently it is in the errored state, because the source repository does not yet exist.

## 12.4 Testing Automated Terraform Workflows

In this section we will test out the workflow for our Terraform CI/CD pipeline. This verifies that the pipeline works and can be used for real deployments.

### 12.4.1 Creating a Source Repository

Create a Terraform workspace in a new directory with one file, `main.tf`. It doesn't really matter what we deploy, because we are just using it to test the pipeline. But I think it's best to pay homage to chapter 1 by reusing the "Hello World!" example. Add the code from Listing 12.14 to `main.tf`:

#### **Listing 12.14 main.tf**

```
provider "aws" {
  region = "us-west-2" #A
}
```

```

data "aws_ami" "ubuntu" {
  most_recent = true

  filter {
    name   = "name"
    values = ["ubuntu/images/hvm-ssd/ubuntu-bionic-18.04-amd64-server-*"]
  }

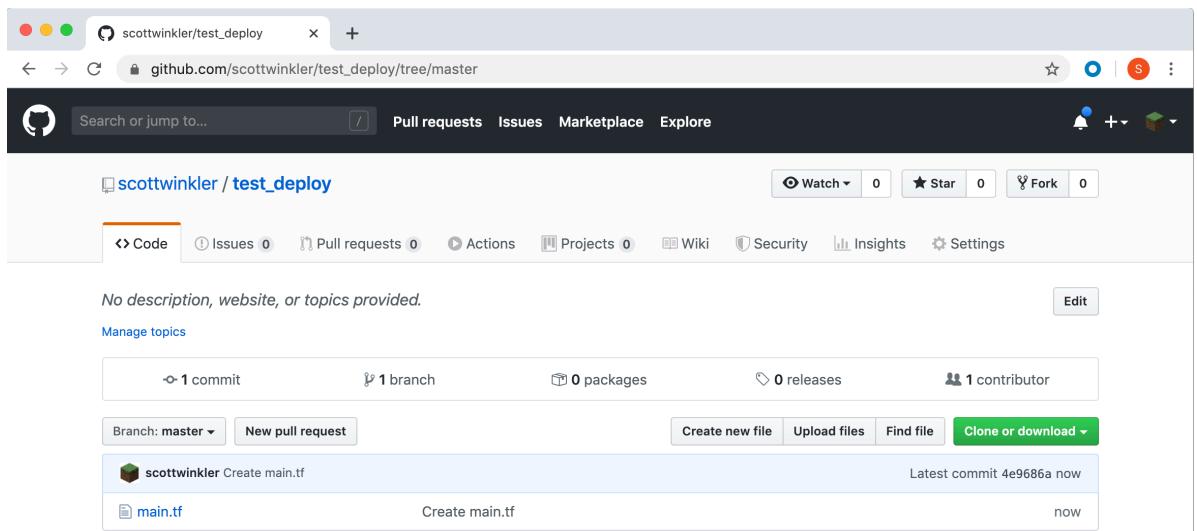
  owners = ["099720109477"]
}

resource "aws_instance" "helloworld" {
  ami = data.aws_ami.ubuntu.id
  instance_type = "t2.micro"
}

```

#A AWS credentials are supplied as environment variables by the pipeline

Now upload this file to a GitHub repository that shares the identifier you specified earlier (see figure 12.10).



**Figure 12.10** A source GitHub repository with the “Hello World!” configuration code. The repository identifier and branch must match the values specified in the previous section.

### How to set Terraform variables in a workspace?

Although we are already using environment variables to pass in AWS credentials, we did not discuss how to set regular Terraform variables in the pipeline, as it isn’t necessary for the “Hello World!” deployment. For any non-trivial deployment, you will need to consider how you would like to set Terraform variables. There isn’t exactly one best way to

do it, and it mostly comes down to personal preference and the amount of work you want to put in. Three strategies, from easiest to hardest, are listed below:

1. **Checking `terraform.tfvars` into version control** – as long as the variables definition file doesn't contain any secrets, it's fine to check this file into GitHub
2. **Pass as an attribute Into the pipeline module** – You can add an attribute of type `map(any)` to the pipeline module. The attribute value may be transformed with `jsonencode()` to produce a JSON string that can be stored as an environment variable on the container runtime. Similar to what we did with `BACKEND`, you can write the environment variable out as a JSON file (`terraform.tfvars.json`) prior to performing `terraform init`.
3. **Dynamically reading variables from a central store** – This is by far the safest and most flexible solution, but also the hardest to administer from a governance perspective. By adding a few lines of code to reach out and download a `terraform.tfvars` file prior to running `terraform init`, you can be sure of exactly which variables were used to run each execution. A centralized store could be implemented as a version controlled S3 bucket.

For the first time through, the webhooks on the GitHub repository will not have been created since we deployed the pipeline before the repo actually existed. There are two ways to resolve this: 1) destroy and recreate the pipeline or 2) click "release change" button in the AWS console. I prefer method #2, which is shown in figure 12.11.

**NOTE** After the webhook is created, you no longer have to click "release change" to manually trigger runs, as it's done automatically whenever a commit is made to the upstream source repository

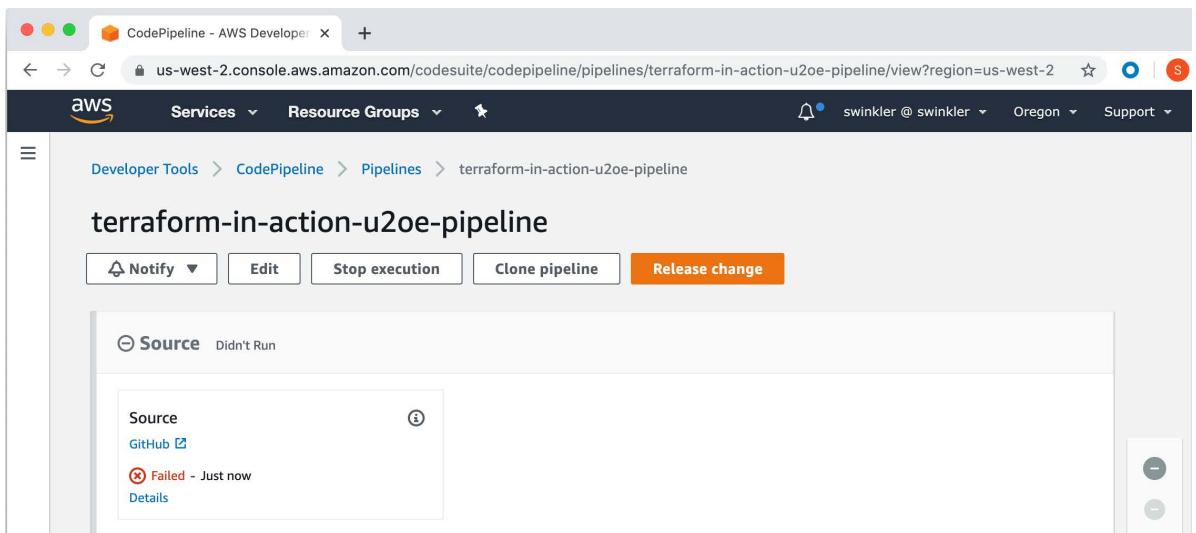


Figure 12.11 click the "release change" button to retry execution and create webhooks on the repo.

After the plan succeeds, you will be prompted to manually approve (figure 12.12). After manual approval is given, the apply will commence and the EC2 instance will be deployed to AWS (figure 12.12).

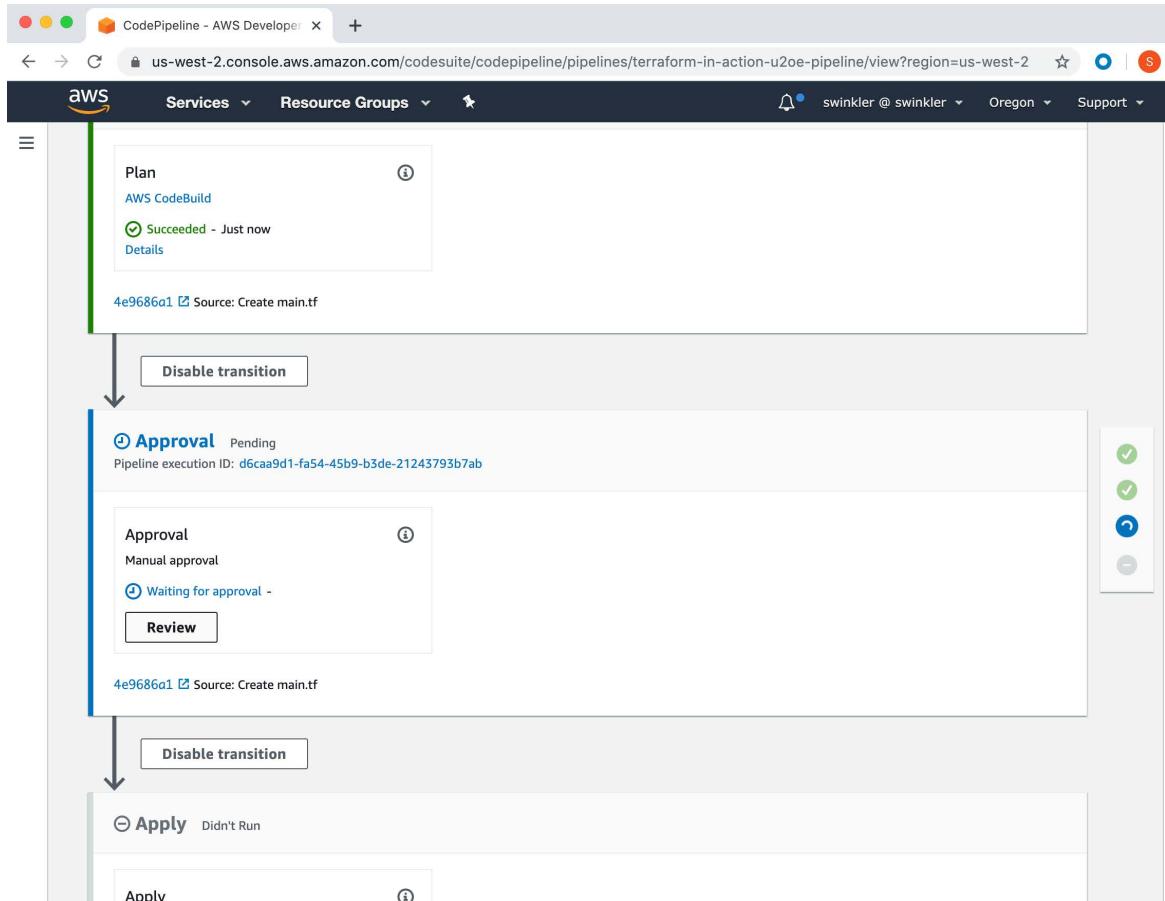


Figure 12.12 After the plan succeeds you will need to give manual approval before the apply will run

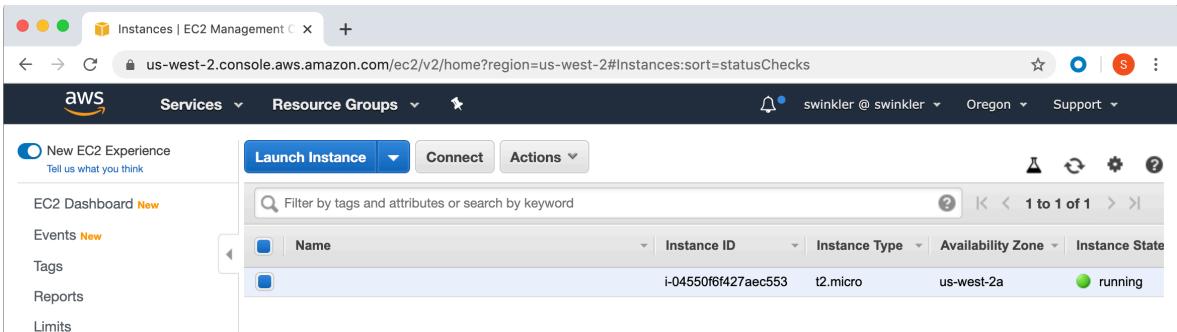


Figure 12.13 Here is the EC2 instance deployed as a result of running Terraform through the pipeline

### 12.4.2 Queuing a Destroy Run

Destroy runs will either be something you perform fairly regularly or almost never. There's not a lot of in-between. For this scenario, I have followed the example of Terraform Enterprise by using a special `CONFIRM_DESTROY` environment variable to trigger destroy runs. If this flag is set to zero, a normal `terraform apply` takes place. If it is set to anything else, a `terraform destroy` runs instead.

Let's queue a destroy plan to clean up the EC2 instance. If we deleted the pipeline without first queuing a destroy run, we would be stuck with orphaned resources (the EC2 instance would still exist, but wouldn't have a state file managing it anymore, as the S3 backend would have been deleted). You will have to update the code of the root module to set the `CONFIRM_DESTROY` to 1. Also set `auto_apply` to `true` at this time, so we don't have to perform a manual approval.

#### **Listing 12.15 main.tf**

```
variable "aws" {
  type = object({access_key = string, secret_key = string, region = string})
}

variable "vcs_repo" {
  type = object({identifier = string, branch = string, oauth_token = string})
}

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

module "s3backend" {
  source = "scottwinkler/s3backend/aws"
}

module "codepipeline" {
  source      = "./modules/codepipeline"
```

```

name          = "terraform-in-action"
vcs_repo      = var.vcs_repo
auto_apply    = true
environment = {
  AWS_ACCESS_KEY_ID      = var.aws.access_key
  AWS_SECRET_ACCESS_KEY = var.aws.secret_key
  CONFIRM_DESTROY        = 1
}
s3_backend_config = module.s3backend.config
}

```

Apply the changes with a `terraform apply`.

```

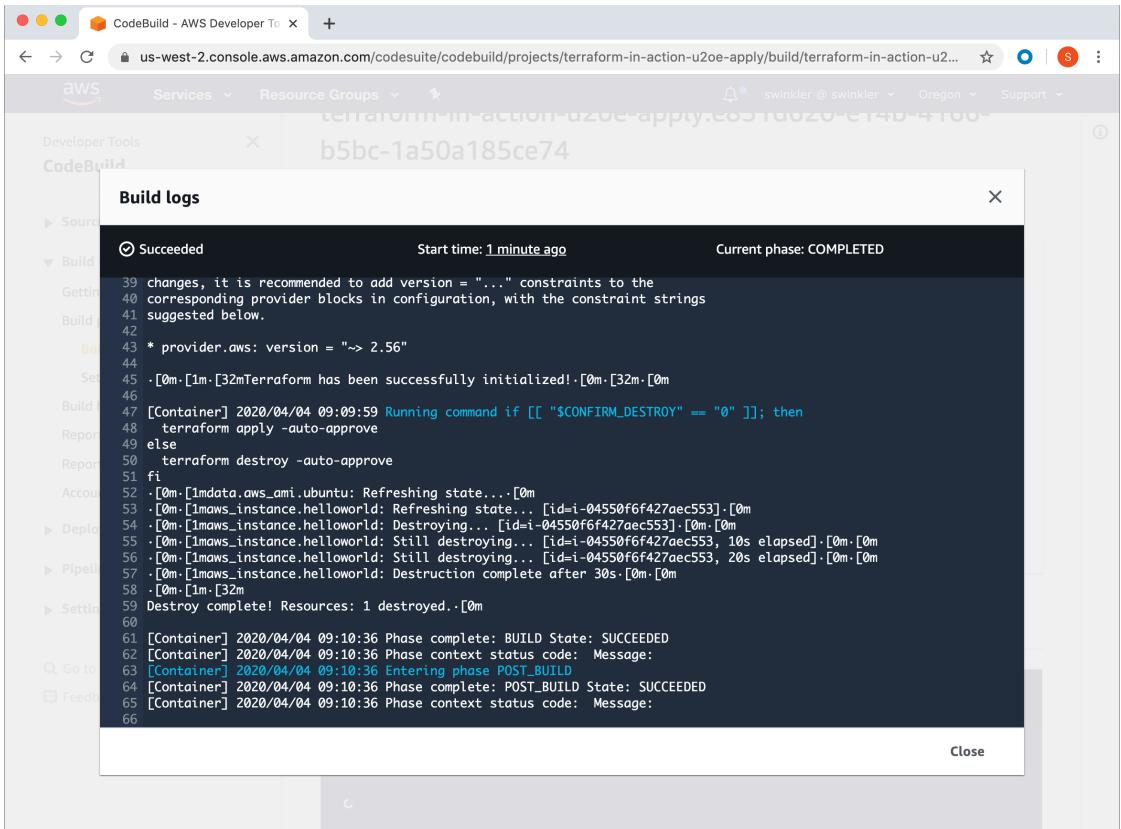
$ terraform apply -auto-approve
...
module.codepipeline.aws_codepipeline.codepipeline: Modifying... [id=terraform-in-action-u2oe-pipeline]
module.codepipeline.aws_codepipeline.codepipeline: Modifications complete after 1s
[id=terraform-in-action-u2oe-pipeline]

Apply complete! Resources: 0 added, 3 changed, 0 destroyed.

```

After the apply succeeds you will need to manually trigger a destroy run by clicking “release change” in the UI (although you won’t have to do a manual approval this time, at least).

**NOTE** since we already have the webhooks in place, the destroy run could also be triggered by pushing a change to GitHub



**Figure 12.14** Logs from AWS CodeBuild after completing a destroy run. The previously provisioned EC2 instance is destroyed

This concludes the Terraform in Automation scenario. Once the EC2 instance has been deleted, you should clean up the pipeline by performing a `terraform destroy`.

```
$ terraform destroy -auto-approve
module.s3backend.aws_kms_key.kms_key: Destruction complete after 23s
module.s3backend.random_string.rand: Destroying... [id=s1061cxz3u3ur7271yv8fgg7]
module.s3backend.random_string.rand: Destruction complete after 0s

Destroy complete! Resources: 18 destroyed.
```

## 12.5 Fireside Chat

In this chapter we developed a CI/CD pipeline for running Terraform in automation. We used a four-stage CI/CD pipeline to download code from a GitHub repository, run a `terraform plan`, wait for manual approval, and perform a `terraform apply`. All of this was to tackle the problem of how to run Terraform at scale from an automation perspective. In the next chapter

we will continue with the meta-Terraform theme, but shift our focus toward secrets management and considering Terraform through the lens of security and governance.

### 12.5.1 FAQ

Before we go, I want to cover some questions I frequently get asked about running Terraform in automation, but didn't have a chance to address earlier in the text:

- **How do I implement a Private Module Registry?**

As you may recall, there are multiple ways to source modules. If the goal is to have the modules not be publicly accessible, you can easily accomplish this by storing modules in a private S3 bucket. Keep in mind that your AWS access keys will need permissions to read from this bucket.

- **How do I install custom and third-party providers?**

Any provider that's in the public provider registry will be downloaded as part of `terraform init`. Custom and third-party providers, must be downloaded and installed some other way *before* running `terraform init`. Where you store these providers is up to you. A simple solution is to bundle them all together into a single versioned zip file and download it at runtime. You can add an input variable to the module for locking to a specific bundle version.

- **How do I handle other kinds of secrets variables and environment variables?**

We will discuss everything you need to know about secrets and secrets management in chapter 12.

- **How do I deploy a project that has multiple environments?**

There are three main strategies for deploying projects that have multiple environments. What you choose really comes down to a matter of personal preference:

- **GitHub Branches** – each logical environment is managed as its own GitHub branch, for example "dev", "staging", and "prod". Promoting from one environment to the next is accomplished by merging a pull request from a lower branch into a higher branch. The advantage of this strategy is that's quick to implement and works well with any number of environments, the disadvantage is that it requires strict adherence to GitHub merging best practices. You don't want someone merging a "dev" branch directly into "prod", because then "staging" would quickly become out of sync and irrelevant.
- **Many-staged pipelines** – As discussed earlier, a Terraform CI/CD pipeline generally has four stages [source, plan, approve, apply], but there is no

reason it has to be four. If you want to deploy multiple environments, you could simply add additional stages to the pipeline for each environment. For example, to deploy to three environments, you could have a ten-stage pipeline: [source, plan (dev), approve (dev), apply (dev), plan(staging), approve (staging), apply (staging), plan(prod), approve(prod), apply(prod)]. I do not like this method because it only works for linear pipelines and does not allow bypassing lower level environments in the event of a hotfix.

- **Linking pipelines together** – This is the most extensible and flexible option of the three, but also the hardest to implement. The idea is simple enough though: a successful apply from one pipeline triggers the execution of the next pipeline. Configuration code is promoted from one pipeline to the next so that only the lowest level environment is connected directly to a version-controlled source repository, the others get their configuration code from earlier environment. The advantage of this strategy is that it allows you to “rollback” individual environments to previously deployed configuration versions, if necessary. Additionally, because each environment is managed as its own pipeline, you can choose to coordinate them all together with your CI tool of choice. This allows you to better integrate with existing workflows and toolchains you may have developed within your organization.

## 12.6 Summary

- Terraform can be run as part of an automated CI/CD pipeline. The reason why you would want to do this, rather than use Terraform Cloud or Terraform Enterprise, is for cost savings and ownership.
- A typical CI/CD pipeline consists of four stages: 1) source 2) plan 3) approve 4) apply. You can tweak the stages to better fit your organization’s needs.
- JSON syntax is favored over HCL when generating configuration code. Although it’s generally more verbose and harder to read than HCL, JSON is more machine friendly and has better library support.
- Dynamic blocks can be toggled on or off with a Boolean flag. This is helpful when you have a code block that needs to exist or not exist depending on the result of a conditional expression.

# 13

## *Secrets Management*

### **This chapter covers:**

- Securing State and Log Files
- Managing Static and Dynamic Secrets
- Enforcing “Policy as Code” with Sentinel

On July 25<sup>th</sup>, 2019, it was reported that the Democratic Senatorial Campaign Committee (DSCC) had exposed over 6.2 million email addresses in what became one of the largest data breaches of all time. The vast majority of addresses came from American consumers, although thousands of university, government and military personnel’s emails were compromised as well. The root cause behind the incident was a small configuration error: an improperly configured S3 bucket. The email addresses had been stored in a single large spreadsheet, named “EmailExcludeClinton.csv”, which was marked as publicly accessible to anyone with an AWS account. At the time of discovery, the data has been left exposed on the Internet for at least nine years.

This little homily should serve as a warning to those who would fail to take information security seriously. Data breaches are enormously detrimental, not only for the public, but for corporations as well. Loss of brand reputation, loss of future revenue, and government-imposed fees and fines are just some of the potential consequences. All it takes for a data breach to occur is a slight oversight, such as an improperly configured S3 bucket that may not even be discovered until many years down the road.

Security is everybody’s responsibility, however, as a Terraform developer, your share of the responsibility is greater than most. Terraform is an infrastructure provisioning technology, and therefore handles a lot of secrets – more than most people realize. Secrets like database passwords, high level access keys, and even private encryption keys may all be consumed and managed by Terraform. Worse, many of these secrets appear as plaintext, in either Terraform

state or log files. Knowing how and where secrets have the potential to be leaked is critical to developing an effective counter strategy. You have to think like the hacker in order to protect yourself from the hacker.

Secrets management is all about how to keep secret information secret. Best practices of secrets management with Terraform, as we will discuss in this chapter, include:

1. Securing State Files
2. Securing Logs
3. Managing Static Secrets
4. Dynamic “Just-in-time” Secrets
5. Enforcing “Policy as Code” with Sentinel

## 13.1 Securing Terraform State

It’s a sad and inevitable truth that sensitive data will find its way into Terraform state, pretty much no matter what you do. Terraform is fundamentally a state management tool, so in order to perform basic execution tasks like drift detection, it needs to compare previous state with current state. Terraform does not treat attributes containing sensitive data any differently than it treats non-sensitive attribute. Therefore, any and all sensitive data gets put in the state file, which is stored as plaintext JSON. Because you can’t prevent secrets from making their way into Terraform state, it’s imperative that you treat the state file itself as sensitive and secure it accordingly.

In this section we will discuss three methods for securing state files:

1. Removing Unnecessary Secrets
2. Least Privileged Access Control
3. Encryption at Rest

### 13.1.1 Removing Unnecessary Secrets

Although you ultimately cannot avoid secrets from wheedling their way into Terraform state, there’s still no excuse for complacency. You should never expose more sensitive information than is absolutely required. If the worst were to happen, and, despite your best efforts and safety precautions, the contents of Terraform state were to be leaked, it is better to expose one secret than a dozen (or a hundred).

**TIP** the fewer secrets you have in Terraform state, the less you have to lose in the event of a data breach

In order to minimize the number of secrets being stored in Terraform state, you first have to know what has the possibility of storing stateful data. Fortunately, this list isn’t too long. There are only three configuration blocks that can store stateful data (sensitive or otherwise) in Terraform. These being: resources, data sources, and output values. Every other kind of configuration block (provider, input variable, local value, module, etc.) do not store stateful

data. Any of these other blocks may still leak sensitive information in other ways, but at least you do not need to worry about them getting anything unwanted in the state file.

Now that you know which blocks have the ability to store sensitive information in Terraform, you have to then determine which secrets are necessary and which are not. A lot of this depends on the level of risk you are willing to take, and how much you'd like to automate with Terraform. An example of a "necessary" secret is shown in snippet 13.1. This code declares an RDS database instance, passing in two secrets, `var.username` and `var.password`. Since both of these attributes are marked as `required`, if you want Terraform to provision and manage an RDS database resource, you have to be willing to accept that your master username and password secret values will exist in Terraform State.

### **Snippet 13.1 rds database source code**

```
resource "aws_db_instance" "database" {
  allocated_storage      = 20
  engine                 = "postgres"
  engine_version         = "9.5"
  instance_class          = "db.t3.medium"
  name                   = "ptfe"
  username               = var.username #A
  password               = var.password #A
}
```

#A username and password are attributes of the `aws_db_instance` resource. These are necessary secrets, because it is impossible to provision this resource without storing the values in Terraform state.

Listing 13.1 shows Terraform state for a deployed RDS instance. Notice that `username` and `password` appear as plaintext.

### **Listing 13.1 aws\_db\_instance in Terraform state**

```
{
  "mode": "managed",
  "type": "aws_db_instance",
  "name": "database",
  "provider": "provider.aws",
  "instances": [
    {
      "schema_version": 1,
      "attributes": {
        //not all attributes are shown
        "password": "hunter2", #A
        "performance_insights_enabled": false,
        "performance_insights_kms_key_id": "",
        "performance_insights_retention_period": 0,
        "port": 5432,
        "publicly_accessible": false,
        "replicas": [],
        "replicate_source_db": "",
        "resource_id": "db-06TUYBMS2HGAY7GKSLTL5H4JEM",
        "s3_import": [],
        "security_group_names": null,
        "skip_final_snapshot": false,
```

```

    "snapshot_identifier": null,
    "status": "available",
    "storage_encrypted": false,
    "storage_type": "gp2",
    "tags": null,
    "timeouts": null,
    "timezone": "",
    "username": "AzureDiamond" #A
}
}
]
}
```

#A username and password are visible as plaintext in terraform.tfstate

Setting secrets on a database instance may be unavoidable, but there are plenty of examples of avoidable situations. One such that comes to mind is passing in the RDS database username and password to a lambda function. Consider the code from listing 13.2, which declares an `aws_lambda_function` resource that has username and password passed from input variables:

### **Listing 13.2 lambda function configuration code**

```

resource "aws_lambda_function" "lambda" {
  filename      = "code.zip"
  function_name = "${local.namespace}-lambda"
  role          = aws_iam_role.lambda.arn
  handler       = "exports.main"

  source_code_hash = filebase64sha256("code.zip")
  runtime        = "nodejs12.x"

  environment {
    variables = {
      USERNAME = var.username #A
      PASSWORD = var.password #B
    }
  }
}
```

#A RDS database username and password

Since the environment block of `aws_lambda_function` contains these values, they will be stored in state just like they were for the database. The difference is that while the RDS database required username and password to be set, the AWS Lambda function does not. The Lambda function only needs credentials to connect to the database instance at runtime (and furthermore, it does not require master username/password, but that is another story).

You might think this is excessive, and possibly redundant. After all, if you are declaring the RDS instance in the same configuration code as your AWS Lambda function, wouldn't the sensitive information be stored in Terraform state regardless? And you would be right. But you would also be exposing yourself to additional vulnerabilities, outside of Terraform. If you aren't

familiar with AWS Lambda, environment variables on Lambda functions are exposed as cleartext to anyone who has read access to that resource (see figure 13.1).

The deployment package of your Lambda function "tia-lambda" is too large to enable inline code editing. However, you can still invoke your function.

Key	Value
PASSWORD	hunter2
USERNAME	admin

**Tags (0)**

A tag is a label that you assign to an AWS resource. Each tag consists of a key and an optional value. You can use tags to search and filter your resources or track your AWS costs.

**Figure 13.1** Environment variables for AWS Lambda functions are visible to anyone with read access in the console. Avoid setting secrets as environment variables in AWS Lambda whenever possible.

Granted, people who have read access to your AWS account are likely to be a small group consisting largely of coworkers and trusted contractors, but do you really want to risk exposing sensitive information in such a way? I would recommend adopting a zero-trust policy, even within your own team.

We can remove `USERNAME` and `PASSWORD` from the environment block by replacing them with a key that tells the Lambda function where to find the secrets. In this case, we will retrieve the sensitive values at runtime from AWS Secrets Manager, a secrets management service that permits rotating, managing and retrieving secrets (Azure and GCP have equivalent services). All we have to do is give the Lambda function permissions to read the secret value from Secrets Manager and we are good to go. Not only does this prevent secrets

from showing up in the state file, it prevents other avenues of sensitive information leakage, such as through the AWS console.

Listing 13.3 shows what the revised configuration code looks like, after replacing database username/password with a secret id that points to where the secret value lives in Secrets Manager.

### Listing 13.3 lambda function configuration code

```
resource "aws_lambda_function" "lambda" {
  filename      = "code.zip"
  function_name = "${local.namespace}-lambda"
  role          = aws_iam_role.lambda.arn
  handler       = "exports.main"

  source_code_hash = filebase64sha256("code.zip")
  runtime        = "nodejs12.x"

  environment {
    variables = {
      SECRET_ID = var.secret_id #A
    }
  }
}
```

#A no more secrets in configuration code! This is just a key for where to fetch the secrets from.

Now, in the application source code, `SECRET_ID` can be used to fetch the secret at runtime (see listing 13.4)

**NOTE** in order for this to work, AWS Lambda needs to be given permission to fetch the secret value from AWS Secrets Manager.

### Listing 13.4 lambda function source code

```
package main

import (
    "context"
    "fmt"
    "os"

    "github.com/aws/aws-lambda-go/lambda"
    "github.com/aws/aws-sdk-go/aws"

    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/secretsmanager"
)

func HandleRequest(ctx context.Context) error {
    client := secretsmanager.New(session.New())
    config := &secretsmanager.GetSecretValueInput{
        SecretId: aws.String(os.Getenv("SECRET_ID")),
    }
    val, err := client.GetSecretValue(config) #A
```

```

    if err != nil {
        return err
    }

    // do something with secret value
    fmt.Printf("Secret is: %s", *val.SecretString)

    return nil
}

func main() {
    lambda.Start(HandleRequest)
}

```

#A fetching the secret dynamically by its id

If you don't understand AWS Secrets Manager just yet, don't worry. We will revisit it later when we talk about managing dynamic secrets in Terraform.

### 13.1.2 Least Privileged Access Control

Removing unnecessary secrets is always a good idea, but it won't prevent your state file from being leaked in the first place. To do that, you need to treat the state file itself as secret and gate who has access to it. After all, you don't want just anyone having access to your state file. Preferably, users should only be able to access states for which they absolutely need access. A *principle of least privilege* should be upheld, which means giving users and service accounts the minimum privileges required to do their job.

In chapter 6 we did exactly this when we created a module for deploying an S3 backend. As part of this module, we restricted access to the S3 bucket to just the account that required access to it. The S3 bucket holds the state files and although we want to give read/write access to some state files, we may not want to give that access to all users. Listing 13.5 shows an example of the policy we created for enabling least privileged access.

#### **Listing 13.5 IAM least privileged policy for S3 backend**

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "",
            "Effect": "Allow",
            "Action": "s3>ListBucket",
            "Resource": "arn:aws:s3:::tia-state-bucket"
        },
        {
            "Sid": "",
            "Effect": "Allow",
            "Action": [
                "s3>PutObject",
                "s3>GetObject"
            ],
            "Resource": "arn:aws:s3:::tia-state-bucket/team1/*" #A
        },
    ]
},
```

```
{
  "Sid": "",
  "Effect": "Allow",
  "Action": [
    "dynamodb:PutItem",
    "dynamodb:GetItem",
    "dynamodb:DeleteItem"
  ],
  "Resource": "arn:aws:dynamodb:us-west-2:215974853022:table/tia-state-lock"
}
}
```

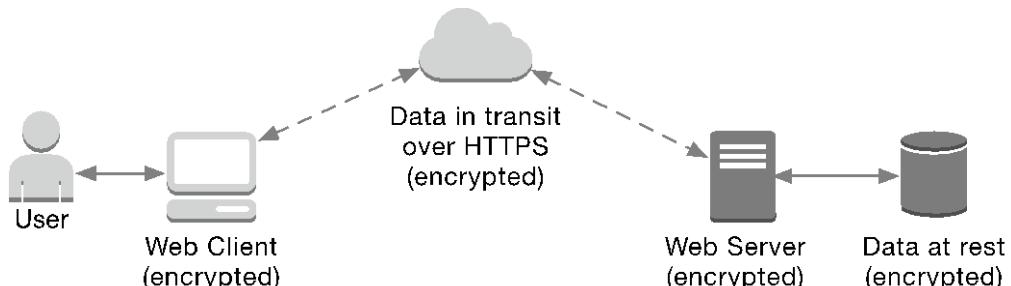
#A this could be further restricted by bucket prefix, if desired

Terraform Cloud and Terraform Enterprise both allow you to restrict user access to state files with their built-in team and organizations system. The basic idea is that users are added to teams, which grant them read/write/admin access to certain workspaces (and their associated state files) in a given organization. For more information about how teams and organizations work, consult the official HashiCorp documentation<sup>1</sup>.

**TIP** besides securing state files, you can also create least privileged IAM roles for users and service accounts deploying Terraform. This is safer than using an admin role, which is what many people do.

### 13.1.3 Encryption at Rest

Encryption at rest is the act of translating data into a format that cannot be decrypted except by authorized users. Even if a malicious user were to gain physical access to the machines storing encrypted data, the data would be useless to them.



<sup>1</sup> <https://terraform.io/docs/cloud/users-teams-organizations/teams.html>

Figure 13.2 Data must be encrypted every step of the way. Most Terraform backends take care of data in transit, but you are responsible for ensuring data is encrypted at rest.

### What about encryption in transit?

Encrypting data in transit is just as important as encryption data at rest. Encrypting data in transit means protecting against network traffic eavesdropping. The standard way this is done is by ensuring data is exclusively transmitted over SSL/TLS, which is enabled by default for most backends, including: S3, Terraform Cloud, and Terraform Enterprise. For some backends, such as the HTTP remote backend, this isn't necessarily true. It is your responsibility to ensure that data is protected both at rest and in transit.

Encryption at rest is easy to enable for most backends. If you are using an S3 backend like the one we created in chapter 6, you can specify a KMS key to use for client-side encryption, or just let S3 use a default encryption key for server-side encryption. If you are using Terraform Enterprise or Terraform Cloud, your data is automatically encrypted at rest by default. In fact, they double encrypt it, once with KMS and again with Vault. For all other remote backends, you will need to consult the relevant documentation to learn how to enable encryption at rest.

### Why not scrub secrets from Terraform state?

There has been much discussion in the community of scrubbing, or removing, secrets from Terraform before they get stored in the state file. One experiment that has been tried is allowing users to provide a PGP key to encrypt sensitive information before it gets stored in Terraform state. This method has been deprecated in newer versions of Terraform, largely because it harder for Terraform to interpolate values. Also, if the PGP key were to be lost (which happens more often than you think), your state file was as good as gone. Nowadays, using a remote backend with encryption at rest enabled is the recommended approach.

## 13.2 Securing Logs

Unsecured log files pose an enormous security risk, and surprisingly, many people aren't even aware of the danger. By reading through the contents of Terraform log files, malicious users can glean sensitive information about your deployment, such as credentials and environment variables, and use them against you. In this section we will discuss some of the ways that sensitive information can be leaked through unsecured logs files, and what you can do to prevent it.

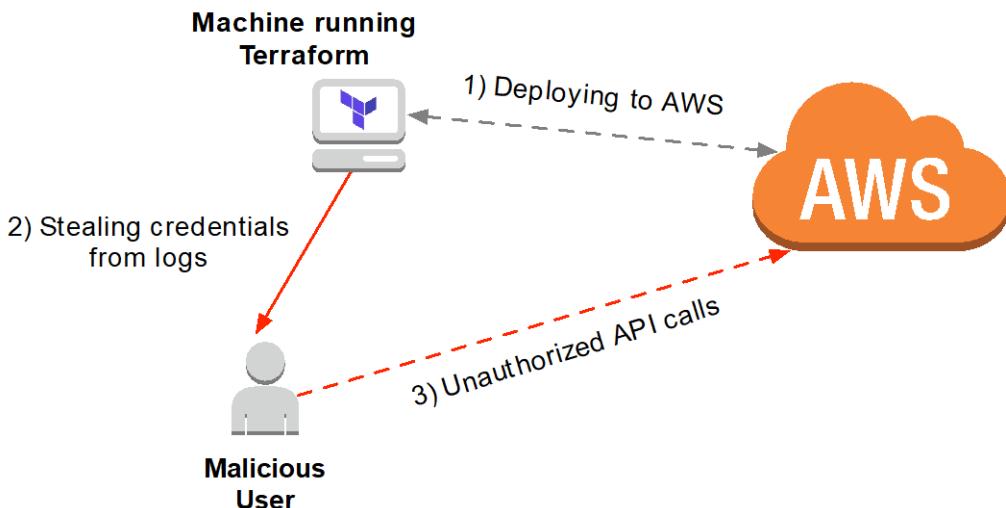


Figure 13.3 a malicious user can steal credentials from log files to make unauthorized API calls on your behalf

### 13.2.1 What Sensitive Information?

Invariably, people are always shocked to learn that sensitive information can appear in log files at all. So much of the official documentation, and online blog articles, focus on the importance of securing the state file, that little is said about the importance of securing logs. I aim to remedy this fallacy. Consider the configuration code in snippet 13.2, which declares a simple “Hello World!” EC2 instance.

#### **Snippet 13.2 “Hello World!” EC2 Instance**

```
resource "aws_instance" "helloworld" {
  ami = var.ami_id
  instance_type = "t2.micro"
  tags = [
    { Name = "HelloWorld" }
  ]
}
```

If you were to create this resource without detailed logging enabled, then little enough of interest would appear in the output logs (see snippet 13.3).

#### **Snippet 13.3 normal log output**

```
$ terraform apply -auto-approve
aws_instance.helloworld: Creating...
aws_instance.helloworld: Still creating... [10s elapsed]
aws_instance.helloworld: Still creating... [20s elapsed]
aws_instance.helloworld: Creation complete after 24s [id=i-002030c2b40edd6bb]
```

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

If, on the other hand, you were to run the same configuration code with trace logs enabled (i.e. `TF_LOG=1`), you would find information about the current caller identity, temporary signed access credentials, and response data from all requests made to deploy the EC2 instance.

Listing 13.6 is a log except when deploying the same configuration code, when having the environment variable `TF_LOG` set to `1`.

### **Listing 13.6 sts:GetCallerIdentity in trace logs**

```
Trying to get account information via sts:GetCallerIdentity
[aws-sdk-go] DEBUG: Request sts/GetCallerIdentity Details:
---[ REQUEST POST-SIGN ]-----
POST / HTTP/1.1
Host: sts.amazonaws.com
User-Agent: aws-sdk-go/1.30.16 (go1.13.7; darwin; amd64) APN/1.0 HashiCorp/1.0
    Terraform/0.12.24 (+https://www.terraform.io)
Content-Length: 43
Authorization: AWS4-HMAC-SHA256 Credential=AKIATESI2XGPMMVVB7XL/20200504/us-east-
    1/sts/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-
    date,
    Signature=c4df301a200eb46d278ce1b6b9ead1cfbe64f045caf9934a14e9b7f8c207c3f8
#A
Content-Type: application/x-www-form-urlencoded; charset=utf-8
X-Amz-Date: 20200504T084221Z
Accept-Encoding: gzip
Action=GetCallerIdentity&Version=2011-06-15
-----
[aws-sdk-go] DEBUG: Response sts/GetCallerIdentity Details:
---[ RESPONSE ]-----
HTTP/1.1 200 OK
Connection: close
Content-Length: 405
Content-Type: text/xml
Date: Mon, 04 May 2020 07:37:21 GMT
X-Amzn-Requestid: 74b2886b-43bc-475c-bda3-846123059142
-----
[aws-sdk-go] <GetCallerIdentityResponse xmlns="https://sts.amazonaws.com/doc/2011-06-15/">
  <GetCallerIdentityResult>
    <Arn>arn:aws:iam::215974853022:user/swinkler</Arn> #B
    <UserId>AIDAJKZ3K7CTQHZ5F4F52</UserId> #B
    <Account>215974853022</Account> #B
  </GetCallerIdentityResult>
  <ResponseMetadata>
    <RequestId>74b2886b-43bc-475c-bda3-846123059142</RequestId>
  </ResponseMetadata>
</GetCallerIdentityResponse>

#A temporary signed credentials that can be used to make a request on your behalf
#B information about current caller identity
```

The temporary signed credentials that appear in the trace logs can be used to make unauthorized requests on your behalf (at least until they expire, which is about 15 minutes).

Listing 13.7 demonstrates using the previous credentials to make a curl request and receiving a response.

#### **Listing 13.7 invoking sts:GetCallerIdentity with signed credentials**

```
$ curl -L -X POST 'https://sts.amazonaws.com' \
-H 'Host: sts.amazonaws.com' \
-H 'Authorization: AWS4-HMAC-SHA256 Credential=AKIATESI2XGPMMVB7XL/20200504/us-east-
    1/sts/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
    Signature=c4df301a200eb46d278ce1b6b9ead1cfbe64f045caf9934a14e9b7f8c207c3f8' \
-H 'Content-Type: application/x-www-form-urlencoded; charset=utf-8' \
-H 'X-Amz-Date: 20200504T084221Z' \
-H 'Accept-Encoding: gzip' \
--data-urlencode 'Action=GetCallerIdentity' \
--data-urlencode 'Version=2011-06-15'

<GetCallerIdentityResponse xmlns="https://sts.amazonaws.com/doc/2011-06-15/">
  <GetCallerIdentityResult>
    <Arn>arn:aws:iam::215974853022:user/swinkler</Arn>
    <UserId>AIDAJKZ3K7CTQHZ5F4F52</UserId>
    <Account>215974853022</Account>
  </GetCallerIdentityResult>
  <ResponseMetadata>
    <RequestId>e6870ff6-a09e-4479-8860-c3ca08b323b5</RequestId>
  </ResponseMetadata>
</GetCallerIdentityResponse>
```

Now, I know what you might be thinking: leaking the results of `sts:GetCallerIdentity` might not be all that bad, considering there is nothing of great importance held therein. But `sts:GetCallerIdentity` is just the beginning. Every single API call that Terraform makes to AWS appears in the trace logs— with the complete request and response objects included. That means that for the “Hello World!” configuration code described above, signed credentials allowing someone to invoke `ec2:CreateInstance` and `vpc:DescribeVpcs` appear as well. Granted, these are temporary credentials that expire in 15 minutes, but risks are risks.

**TIP** always turn off trace logging except when actively debugging

#### **13.2.2 Dangers of Local-Exec Provisioners**

In chapter 7 we introduced `local-exec` provisioners and how they can be used to run commands on a local machine during a `terraform apply` or `destroy`. As previously mentioned, `local-exec` provisioners are inherently dangerous and should be avoided whenever possible. Now, I will give you one more to be leery of them: even when trace logging is disabled, `local-exec` provisioners can be used to print secrets in the log files:

Consider snippet 13.4 which declares a `null_resource`, with an attached `local-exec` provisioner:

#### **Snippet 13.4 null\_resource with a local-exec provisioner**

```
resource "null_resource" "uh_oh" {
```

```

provisioner "local-exec" {
  command = <<-EOF
    echo "access_key=$AWS_ACCESS_KEY_ID"
    echo "secret_key=$AWS_SECRET_ACCESS_KEY"
  EOF
}
}

```

If you were to run this, you would see the following during a terraform apply (even with trace logging disabled):

#### **Snippet 13.5 printing AWS access keys during an apply**

```

$ terraform apply -auto-approve
null_resource.uh_oh: Creating...
null_resource.uh_oh: Provisioning with 'local-exec'...
null_resource.uh_oh (local-exec): Executing: ["./bin/sh" "-c" "echo
  \"access_key=$AWS_ACCESS_KEY_ID\"\necho \"secret_key=$AWS_SECRET_ACCESS_KEY\"\n"]
null_resource.uh_oh (local-exec): access_key=ASIAQHJM6YXTDSEUEMUJ #A
null_resource.uh_oh (local-exec): secret_key=ILjkhTbflyPdxkvWJ19NV8qZXPJ+yVM3JSq3Uaz1 #B
null_resource.uh_oh: Creation complete after 0s [id=5973892021553480485]

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

```

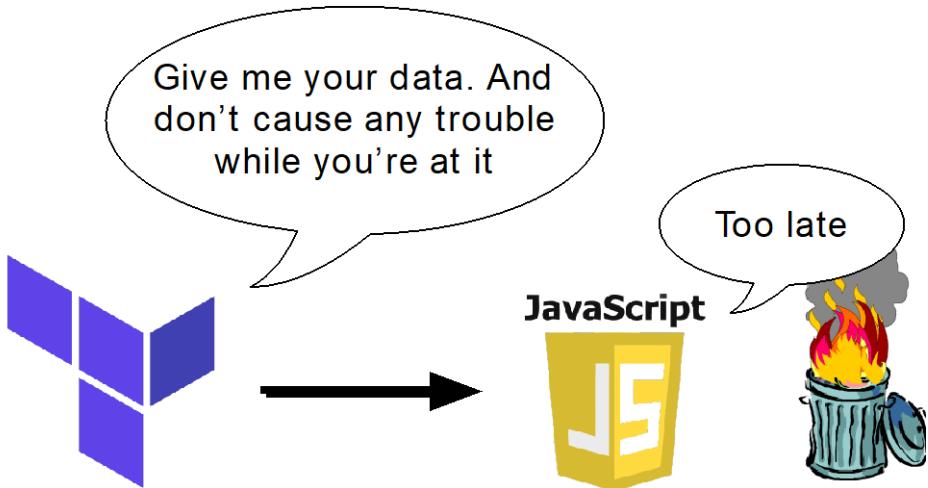
#A AWS Access Key

#B AWS Secret Access Key

**NOTE** AWS access keys are not the only thing that `local-exec` provisioners can expose. Any secrets stored in the filesystem or runtime environment of the machine running Terraform are also vulnerable.

#### **13.2.3 Dangers of External Data Sources**

Somewhat related to `local-exec` provisioners are external data sources. In case you aren't aware of this particular dodgy character, external data sources allow you to execute arbitrary code and return the results back to Terraform. That sounds great at first, because it allows you to create custom data sources without resorting to writing your own Terraform provider. The downside is that *any* arbitrary code can be called, which could be extremely troublesome if you are not careful.



## External Data Source

Figure 13.4 External data sources execute arbitrary code (such as Python, JavaScript, Bash, etc.) and return the results back to Terraform. If the code is malicious, it could cause all sorts of problems before you have a chance to do anything about it.

**TIP** in case you are interested in creating custom resources without writing your own provider I recommend looking into the Shell provider for Terraform<sup>2</sup> (which is also covered in Appendix A)

External data sources are particularly nefarious because they run during a terraform plan, which means that all a malicious user would need to do is sneak this code into your configuration code and make sure it gets run during a terraform plan in order to gain access to all of your secrets. No manual confirmation of an apply necessary.

**TIP** always skim through any module you'd like to use, even if it comes from the official module registry, to ensure that no malicious code is present

Consider this code from snippet 13.6, which doesn't even look that bad at first glance:

### Snippet 13.6 external data source

```
data "external" "do_bad_stuff" {
  program = ["node", "${path.module}/run.js"]
}
```

<sup>2</sup> <https://github.com/scottwinkler/terraform-provider-shell>

However, during a `terraform plan`, this data source will run a Node.js script to execute malicious code. Snippet 13.7 shows an example of what the external script might do:

### Snippet 13.7 nodejs script

```
// runKeyLogger()
// stealBankingInformation()
// emailNigerianPrince()
console.log(JSON.stringify({
    AWS_ACCESS_KEY_ID: process.env.AWS_ACCESS_KEY_ID,
    AWS_SECRET_ACCESS_KEY: process.env.AWS_SECRET_ACCESS_KEY,
}))
```

When this code runs, it can do anything from installing unwanted software on your computer, to stealing your data, or even mining Bitcoins. In this case, the code just returns a JSON object with the AWS access and secret access keys (which is still nasty!). If you were to run this, nothing of interest would show up in the logs:

```
$ terraform apply -auto-approve
data.external.do_bad_stuff: Refreshing state...

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

But in your state file the data would appear in plaintext:

```
$ terraform state show data.external.do_bad_stuff
# data.external.do_bad_stuff:
data "external" "do_bad_stuff" {
  id      = "-"
  program = [
    "node",
    "./run.js",
  ]
  result  = {
    "AWS_ACCESS_KEY_ID"      = "ASIAQHUM6YXTDSEUEMUJ"
    "AWS_SECRET_ACCESS_KEY" = "ILjkhTbflyPdxkvWJl9NV8qZXPJ+yVM3JSq3Uaz1"
  }
}
```

That's not even all of it. The sensitive information also show up in the log files (as long as trace logging is enabled):

```
JSON output: [123 34 65 87 83 95 65 67 67 69 83 83 95 75 69 89 95 73 68 34 58 34 65 83 73 65
  81 72 85 77 54 89 88 84 68 83 69 85 69 77 85 74 34 44 34 65 87 83 95 83 69 67 82 69 84
  95 65 67 67 69 83 83 95 75 69 89 34 58 34 73 76 106 107 104 84 98 102 108 121 80 100
  120 107 118 87 74 108 57 78 86 56 113 90 88 80 74 43 121 86 77 51 74 83 113 51 85 97
  122 49 34 125 10]
```

Converting this byte array to a string yields the following JSON string:

```
{
  "AWS_ACCESS_KEY_ID": "ASIAQHUM6YXTDSEUEMUJ",
  "AWS_SECRET_ACCESS_KEY": "ILjkhTbflyPdxkvWJl9NV8qZXPJ+yVM3JSq3Uaz1"
```

}

**NOTE** External data sources are perhaps the most dangerous resource in all of Terraform. Be extremely judicious with their use, as there are many clever and devious ways that sensitive information can be leaked with them.

### 13.2.4 Restricting Access to Logs

A lot of the same rules for securing state files apply to log files, namely A) You don't want people reading log files if it's not required to do their job, and B) You want to encrypt data at rest and in transit so that there is no possibility of hackers or eavesdroppers gaining access to your data.

Here are some additional guidelines specific to securing log files:

1. Do not allow unauthorized users to run plans or applies against your workspace
2. Turn trace logging off except when actively debugging
3. If you have continuous integration webhooks setup on a repository, do not allow `terraform plan` to be run for pull requests initiated from forks. This would allow hackers to run external data sources without your explicit approval.

**TIP** Relax, I'm not trying to scare you. Only a handful of Terraform users are advanced enough to know about any of these exploits, and of those who do, few have reason to cause you harm. Just use your best judgement and don't take any unnecessary risks.

## 13.3 Managing Static Secrets

Static secrets are sensitive values that do not change, or at least not often. Most secrets could be classified as static secrets. Things like username and passwords, long-lived OAuth tokens, and environment variables or config files with credentials in them are all examples of static secrets. In this section we will discuss some of the different ways to manage static secrets, as well as an overview of how to enact an effective secrets rotation policy.

### 13.3.1 Environment Variables

There are two major ways to pass static secrets into Terraform: A) as environment variables and b) as Terraform variables. I recommend passing secrets in as environment variables whenever possible because it is far safer than the alternative. For one, environment variables do not show up in the state or plan files, and for two it's harder for malicious users to access your sensitive values as compared to Terraform variables. In the previous section we discussed how environment variables can be leaked with `local-exec` provisioners and external data sources, but both of these risks can be mitigated with either careful code reviews or Sentinel policies (as we shall see in section 13.5).

As safe as environment variables tend to be, they, with few exceptions, are only able to configure secrets in Terraform providers. Some rare resources have the ability to read

variables from the environment as well, but you would know which ones these are if you happen to come across one.

When configuring a Terraform provider, you definitely do not want to pass sensitive information in as regular Terraform variables (see snippet 13.8).

### **Snippet 13.8 configuring provider with Terraform variables**

```
provider "aws" {
  region      = "us-west-2"
  access_key  = var.access_key #A
  secret_key  = var.secret_key #A
}

#A very bad idea!
```

I know I've done this a couple time in previous examples throughout the book, but that's because I try to be as concise as possible, and sometimes that means sacrificing security best practices. In general, configuring sensitive information in providers with Terraform variables is dangerous. The reason why it is so bad is because it opens you up to the possibility of someone redirecting secrets and using them elsewhere. Consider how easy it is for someone to output the AWS access and secret access keys by adding simply adding the following lines to the configuration code:

### **Snippet 13.9 outputting Terraform variables**

```
output "aws" {
  value = {
    access_key = var.access_key,
    secret_key = var.secret_key
  }
}
```

Another possibility is saving the contents to a `local_file` resource, such as in Snippet 13.10:

### **Snippet 13.10 local\_file to save access and secret access keys**

```
resource "local_file" "aws" {
  filename = "credentials.txt"
  content = <<-EOF
  access_key = ${var.access_key}
  secret_key = ${var.secret_key}
  EOF
}
```

Or even uploading to an S3 bucket:

### **Snippet 13.11 uploading credentials to an S3 bucket**

```
resource "aws_s3_bucket_object" "aws" {
  key      = "creds.txt"
  bucket   = var.bucket_name
  content = <<-EOF
```

```

access_key = ${var.access_key}
secret_key = ${var.secret_key}
EOF
}

```

As you can see, it doesn't take a genius to be able to read sensitive information from Terraform variables. Furthermore, the avenues of potential attack are more numerous compared to environment variables, so it's nearly impossible to develop an effective governance approach or counter strategy. Anyone with access to modify your configuration code or run plans/applies on your workspace can easily steal secret values. Obviously, the biggest threat will be people that are closest to you, such as coworkers and contractors. But you don't want anyone having the ability to steal sensitive information, even if they are only stealing secrets to bypass red tape and make their job easier (I am guilty of this).

### **Snippet 13.12 configuring provider with environment variables**

```

provider "aws" {
  region      = "us-west-2" #A
}

```

#A access key and secret key are set as environment variables instead of as Terraform variables

Some providers allow you to set secret information in other ways, such as through config file. This works fine for most business use cases. It can be a little awkward when running Terraform in automation, but it's no big deal. What you should be aware of is that nothing on your machine is truly secret, config files included. Consider the code from snippet 13.13 which declares a `local_file` data source to read data from an AWS credentials file:

### **Snippet 13.13 reading the AWS credentials file**

```

data "local_file" "credentials" {
  filename = "/Users/Admin/.aws/credentials"
}

```

I know that this is a bit of a contrived example, and I doubt you will ever encounter this exact situation yourself, but it is something to be aware of, nonetheless. Just because a file is marked as hidden on your filesystem doesn't mean Terraform can't access it.

**WARNING!** malicious Terraform code can access any secret stored anywhere on the local machine running Terraform!



Figure 13.5 No secret is safe from the prying eyes of Terraform

### 13.3.2 Terraform Variables

Despite all the shortcomings of Terraform variables, sometimes you do not have a choice in the matter. Recall the database instance we declared earlier:

#### **Snippet 13.14 rds database source code**

```
resource "aws_db_instance" "database" {
  allocated_storage      = 20
  engine                 = "postgres"
  engine_version         = "9.5"
  instance_class          = "db.t3.medium"
  name                   = "ptfe"
  username               = var.username #A
  password               = var.password #A
}
```

#A cannot be set from environment variables

If you wish to deploy this resource, you are stuck setting username and password as Terraform variables, as there is no option for using environment variables.

For this case, you can still use Terraform variables to set sensitive information as long as you are smart about it. Firstly, I recommend running Terraform in automation, if you are not already. It is imperative that a single source of truth is maintained for configuration code. You do not want people deploying Terraform from their local machines, even if they are using a remote backend like S3. By ensuring that Terraform runs are always linked to a Git commit, troublemakers will not be able to insert malicious code without leaving behind incriminating evidence in the Git history.

After you are running Terraform in automation, you should seek to isolate sensitive Terraform variables from non-sensitive Terraform variables. Terraform Cloud and Terraform Enterprise make this easy, because they let you mark specific variables as sensitive when creating through either the UI or API. Figure 13.6 shows this in action.

### Terraform Variables

These [Terraform variables](#) are set using a `terraform.tfvars` file. To use interpolation or set a non-string value for a variable, click its HCL checkbox.

Key	Value
access_key	<input type="text" value="value"/> <input checked="" type="checkbox"/> HCL <input type="checkbox"/> Sensitive
Description	<input type="text" value="description (optional)"/>
<input type="button" value="Save variable"/> <input type="button" value="Cancel"/>	

### Environment Variables

These variables are set in Terraform's shell environment using `export`.

Key	Value
There are no variables set.	
<input type="button" value="Add variable"/>	

**Figure 13.6 Terraform variables can be marked as sensitive by clicking the “Sensitive” checkbox**

If you aren't using Terraform Cloud/Terraform Enterprise, you will have to segregate sensitive Terraform variables yourself. One way to accomplish this is by deploying workspaces with multiple variables definitions files. Terraform does not automatically load variables definitions file with any name other than `terraform.tfvars`, but you can specify other files by using the `-var-file` flag. For instance, if you can have non-sensitive data stored in `production.tfvars` (checked into Git) and sensitive data stored in `secrets.tfvars` (not checked into Git), the following command would do the trick:

#### Snippet 13.15 deploying with two variables definition files

```
$ terraform apply \
-var-file="secrets.tfvars" \
-var-file="production.tfvars"
```

## 13.4 Utilizing Dynamic Secrets

Secrets should be rotated periodically, at least once every 90 days, or in response to potential leaks. You don't want people stealing secrets and using them for something else, so the smaller the window of time a secret is valid for, the better. Ideally, secrets should not even

exist until they are needed (i.e. leased or created "just-in-time") and they should be revoked immediately after use. Secrets like these are called *dynamic secrets*, and they are substantially more secure than static secrets.

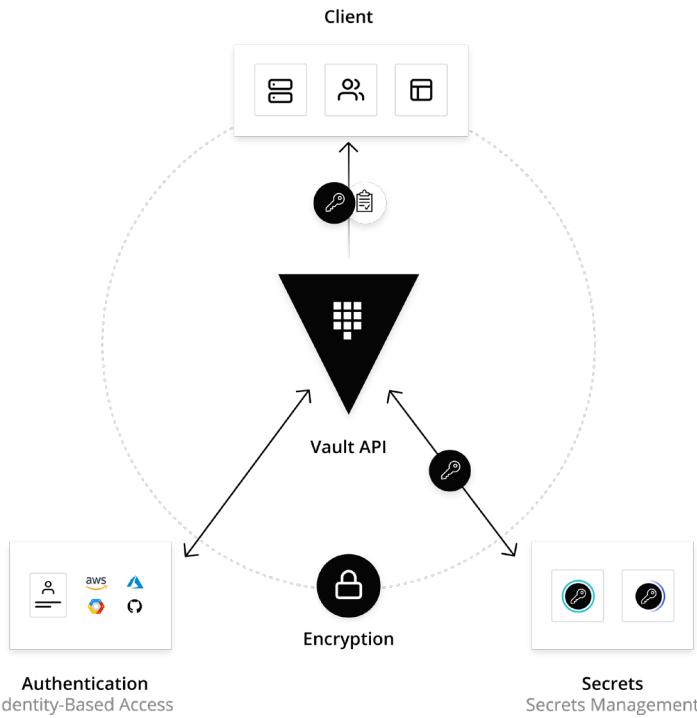
We briefly mentioned dynamic secrets earlier, when we discussed the importance of removing unnecessary secrets from Terraform. That was about moving secrets out of Terraform configuration and into the application layer. For dynamic secrets that cannot be moved into the application layer, the recommended approach is to use a data source that can read secrets from a secrets provider during a Terraform execution.

**NOTE** If you are running in a custom automation environment, you could also write your own logic for reading dynamic secrets, that does not involve data sources

In this section we will discuss how data sources from secrets provider like HashiCorp Vault or AWS Secrets Manager can be used to dynamically read secrets into Terraform variables.

### 13.4.1 HashiCorp Vault

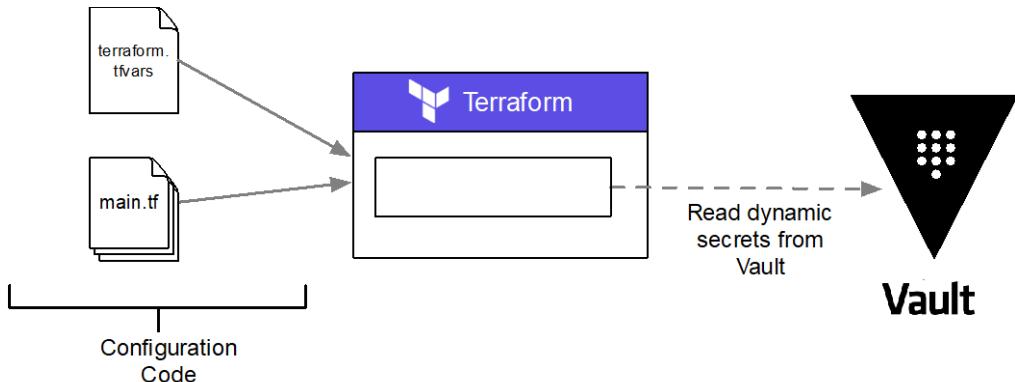
HashiCorp Vault is a secrets management tool designed to allow you to centralize your secrets, authenticate clients against a wide variety of identity providers, and introduce fine grained access controls and audit logging for zero-trust networks. It's a great tool for managing static and dynamic secrets and is fast becoming a standard in the information security industry. Currently, Vault is HashiCorp's biggest source of revenue, with over a hundred million U.S. dollars in revenue, as of 2020.



**Figure 13.7** Vault is a secrets management tool that allows you to store, access, and distribute secrets by authenticating clients against various identity providers

The operationalization and deployment of Vault is outside the scope of this book, but there are many resources on the HashiCorp Learn site, and modules in the module registry, that can help you get started. What we will talk about is how Terraform can integrate with Vault to read dynamic secrets at runtime.

Vault exposes an API for creating, reading, updating and deleting secrets. As you might expect, this also means that there's a Vault provider for Terraform that allows it to manage Vault resources. The Vault provider for Terraform is no different than any other provider; you declare what you want in code and Terraform takes care of making the backend API calls on your behalf (see figure 13.8).



**Figure 13.8** the Vault provider works just like any other Terraform provider; it integrates with the API backend and exposes resources and data sources to Terraform. Some of these data sources may be used to read dynamic secrets at runtime.

Sample code for configuring the Vault provider, reading secrets from a data source, and using these secrets to configure the AWS provider is shown in Listing 13.8. Every time Terraform runs, new short-lived access credentials will be obtained from Vault.

**WARNING!** All the previous rules still apply! You still have to securely manage Terraform variables, state files, and log file.

#### Listing 13.8 configuring Terraform with Vault

```
provider "vault" {
  address = var.vault_address
}

data "vault_aws_access_credentials" "creds" {
  backend = "aws"
  role    = "prod-role"
}

provider "aws" {
  access_key = data.vault_aws_access_credentials.creds.access_key
  secret_key = data.vault_aws_access_credentials.creds.secret_key
  region     = "us-west-2"
}
```

**NOTE** to reduce the risk of secrets exposure, the Vault provider requests tokens with a relatively short TTL (20 minutes, by default). Any issued credentials will be revoked when the token expires.

#### 13.4.2 AWS Secrets Manager

AWS Secrets Manager (ASM) is a competitor to HashiCorp Vault. It allows basic K/V storage and rotation of secrets but is generally less sophisticated than Vault and lacks many of Vault's

more advanced features. The main advantage of ASM is that it's a managed service, which means you don't need to stand up your own infrastructure to use it, it's ready to go right out of the box.

**NOTE** Azure and GCP both have services comparable to ASM, and the process of using them is basically the same

Like Vault, ASM allows you to read dynamic secrets at runtime with the help of data sources. Some sample code for doing this is shown in Listing 13.9.

#### Listing 13.9 configuring Terraform with AWS Secrets Manager

```
data "aws_secretsmanager_secret_version" "db" {
  secret_id = var.secret_id
}

locals {
  creds = jsondecode(data.aws_secretsmanager_secret_version.db.secret_string)
}

resource "aws_db_instance" "database" {
  allocated_storage = 20
  engine           = "postgres"
  engine_version   = "12.2"
  instance_class   = "db.t2.micro"
  name             = "ptfe"
  username         = local.creds["username"]
  password         = local.creds["password"]
}
```

**TIP** if you are not already using Vault for secrets management, AWS Secrets Manager is a great alternative.

## 13.5 Sentinel and Policy as Code

Sentinel is an embeddable policy as code framework designed to automate governance, security, and compliance-based decisions. Complex legal and business requirements, which have traditionally been enforced manually by humans, can be expressed entirely as code with Sentinel policies. Sentinel can automatically prevent out of compliant Terraform runs from executing. For example, you normally do not want someone deploying 5,000 virtual machines without explicit authorization. With Terraform, there are no guardrails to prevent someone from deploying 5,000 virtual machines. The advantage of Sentinel is that you can write a policy to automatically reject such requests, before Terraform applies the changes.

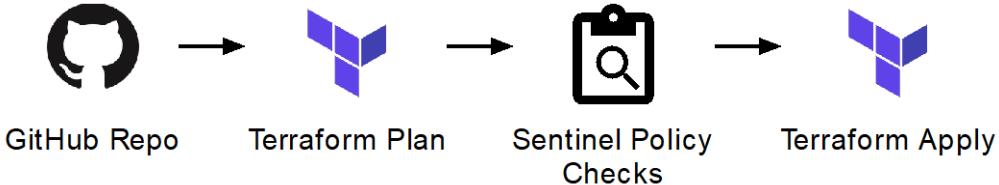


Figure 13.9 Sentinel policies are checked between the plan and apply of a Terraform CI/CD pipeline. If any Sentinel policy were to fail, the run exits with an error condition, and the apply stage is skipped

### History of Sentinel

Sentinel was initially released without much fanfare on September 19<sup>th</sup>, 2017, no doubt the result of a caffeine fueled all-nighter by software genius, and HashiCorp co-founder, Mitchell Hashimoto. At the time, it was not clear how Sentinel could be productized, so nothing much came of it until a few months later, when it was advertised as a premium service offering for HashiCorp Enterprise product lines. At that point Sentinel was still pretty immature as a technology, and I do not know anybody that was actually using it. It remained the forgotten stepchild of HashiCorp for nearly three years.

Nothing much changed because, for the longest time, only a single engineer was assigned to maintaining Sentinel. Now, a team of around 10-20 engineers is dedicated to the technology, with the goal of continuously making improvements and increasing adoption. In March of 2020, version 0.15.0 was released, with significant improvements to the language and technology, that finally convinced me of Sentinel's strong future.

Sentinel is a stand-alone HashiCorp product designed to work with all of HashiCorp's Enterprise service offerings, including: Consul, Nomad, Terraform, and Vault. It has matured over the years and finally found its place under HashiCorp's "Better Together" narrative, which is the catchphrase HashiCorp says to suggest that although their products could be used independently, they complement each other and work best when used together. Before I get you too excited about Sentinel, and the great things it can do, you should know that Sentinel isn't designed to work with open source Terraform, and is itself not available in the open source.

### Can I use Sentinel without an Enterprise License?

Sentinel is distributed as a golang binary, which means that anyone can download it and use it for free (although the source code is still kept secret). The problem is that to do anything useful with Sentinel you still need to have access to the plugins written for Terraform, which are currently reserved only for Enterprise customers (and to a limited extent, Terraform Cloud customers). Sentinel plugins are basically Terraform providers, so it's theoretically possible that someone could write their own plugin that has all the same features as the one HashiCorp created, and open source it. So far nobody has taken the initiative, but if this were to be done, then anyone could use Sentinel with Terraform and not have to pay HashiCorp anything. I imagine if Sentinel ever becomes more popular, someone will end up doing this, and it will be quite interesting to see how HashiCorp handles the situation.

### 13.5.1 Writing a Basic Sentinel Policy

Sentinel policies are not written in HCL, as you might expect, instead they are written in *Sentinel*. *Sentinel* is its own domain specific programming language, which has a passing resemblance to Python. Sentinel policies are made up of rules, which are basically just functions that return either true or false (pass or fail). As long as all rules in a policy pass, the policy itself passes. If you are using Sentinel in a CI/CD pipeline, that means execution will continue to the next stage.

A trivial Sentinel policy, which passes for all use cases, is shown in Snippet 13.16.

#### Snippet 13.16 trivial Sentinel policy

```
main = rule { #A
    true
}

#A policy with a single rule that always evaluates to true
```

#### Why not Python, Ruby, or some other programming language?

Sometimes I think that Mitchell Hashimoto and Armon Dadgar (other co-founder of HashiCorp) just like creating new programming languages for the hell of it. After all, why create HCL when JSON or YAML would do? Why create Sentinel when Python or Ruby have been around for years, and have robust testing frameworks of their own? The answer is that Armon and Mitchell are mad geniuses with an ambitious and unwavering vision in mind. They decided that the best way to realize their vision was to create a new programming language.

The most important design element of Sentinel is that it's a sandbox programming language. Most other languages have security loopholes or backdoors that can be used to bypass normal operations and gain higher level system access. Ruby and Python, for example, are both dynamic languages that can be monkey patched at runtime. As a language designed with governance and compliance in mind, Sentinel had to be embeddable so as that be secure from hackers. Other sandbox programming languages like Lua or JavaScript could have worked out too, but the syntax wouldn't have been as clean, as neither was initially designed with the goal of writing policy as code.

As an emerging technology, Sentinel is not as mature as most other programming languages, but it does have all the basic expressions and syntax elements you would come to expect from a modern programming language. It also has an adequate, if rather small, standard library. This makes Sentinel good enough for day-to-day work, even if it's not the most popular programming language ever.

### 13.5.2 Blocking Local-Exec Provisioners

The goal of this book isn't to teach Sentinel, but I do want to give you a feel for what practical problems you can solve with Sentinel. Consider the dilemma we had earlier with being able to print environment variables, such as `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY`, using local-exec provisioners. Snippet 13.17 shows the code that did this:

### Snippet 13.17 null\_resource with a local-exec provisioner

```
resource "null_resource" "uh_oh" {
  provisioner "local-exec" {
    command = <<-EOF
      echo "access_key=$AWS_ACCESS_KEY_ID"
      echo "secret_key=$AWS_SECRET_ACCESS_KEY"
    EOF
  }
}
```

Without Sentinel, you would have to manually skim through all the configuration code to make sure that nobody is abusing `local-exec` provisioners in this way. With Sentinel, you could write a policy to automatically block all Terraform runs which contain configuration code that has the keyword `AWS_ACCESS_KEY_ID` or `AWS_SECRET_ACCESS_KEY` in a provisioner.

Consider the following Sentinel policy which does this:

### Listing 13.10 Sentinel policy for validating local-exec provisioners

```
import "tfconfig/v2" as tfconfig

keywordInProvisioners = func(s){
  bad_provisioners = filter tfconfig.provisioners as _, p {
    p.type is "local-exec" and
    p.config.command["constant_value"] matches s
  }
  return length(bad_provisioners) > 0
}

no_access_keys = rule {
  not keywordInProvisioners("AWS_ACCESS_KEY_ID")
}

no_secret_keys = rule {
  not keywordInProvisioners("AWS_SECRET_ACCESS_KEY")
}

main = rule { #A
  no_access_keys and
  no_secret_keys
}
```

#A this is the rule that disallows access keys and secret keys from being printed by local-exec provisioners

**NOTE** Sentinel policies are not easy to write! You should expect a high learning curve, even if you are already a skilled programmer.

If we incorporate this Sentinel policy as part of our CI/CD pipeline, a subsequent run would fail with the following error message:

```
$ sentinel apply p.sentinel
```

```
Fail
```

**Execution trace.** The information below will show the values of all the rules evaluated and their intermediate boolean expressions. Note that some boolean expressions may be missing if short-circuit logic was taken.

```
FALSE - p.sentinel:19:1 - Rule "main" #A
  FALSE - p.sentinel:20:2 - no_access_keys
    FALSE - p.sentinel:12:2 - not keywordInProvisioners("AWS ACCESS KEY ID")
      TRUE - p.sentinel:5:3 - p.type is "local-exec"
      TRUE - p.sentinel:6:3 - p.config.command["constant value"] matches s

  FALSE - p.sentinel:11:1 - Rule "no_access_keys"
    TRUE - p.sentinel:5:3 - p.type is "local-exec"
    TRUE - p.sentinel:6:3 - p.config.command["constant_value"] matches s
```

#A the main rule has failed because the “no\_access\_keys” composition rule has failed

You can use Sentinel to enforce that any attribute on any resource is what you want it to be. Examples of other common policies include disallowing 0.0.0.0/0 CIDR blocks, restricting instance types of EC2 instances, and enforcing tagging on resources.

**TIP** If you are not a programmer, or don't have time to write your own policies, you can always use those written by other people (i.e. Sentinel modules) instead

## 13.6 Final Words

We are at the end of the last chapter of the book. You finally know not only the fundamentals of Terraform, which is important as an individual contributor, but also how to manage, extend, automate, and secure Terraform. You know all the tricks and backdoors that hackers can use to steal your sensitive information, and more importantly, you know how to fight back. At this point you should feel extremely confident in your abilities to tackle any problem with Terraform. You are the Terraform Guru now. Beware, because now people will look to you for guidance as the resident Terraform expert. Even though this is the end of our journey together, I hope you will have many more great experiences working with Terraform in the future. Thank you for reading.

## 13.7 Summary

- State files can be secured by 1) removing unnecessary secrets 2) least privileged access control 3) encryption at rest
- Log files can be secured by 1) turning off trace logs 2) avoiding the use of local-exec provisioners and external data source
- Static secrets should be set as environment variables whenever possible. If you absolutely must use Terraform variables, consider maintaining a separate `secrets.tfvars` file for this purpose
- Dynamic secrets are far safer than static secrets, being created on demand and valid

for only the period of time they will be used. You can read dynamic secrets with the corresponding data source from the Vault or AWS provider.

- Sentinel can enforce “Policy as Code”. Sentinel policies can automatically reject Terraform runs based on the contents of the configuration code, or the results of a plan.

# Appendix A

## *Creating Custom Resources with the Shell Provider*

**This appendix covers:**

- Installing community Terraform providers from the provider Registry
- Creating a custom data source with the Shell provider
- Creating a custom resource with the Shell provider

The Shell provider<sup>1</sup> is a third-party provider proudly developed and maintained by me. This provider is available on the provider registry and allows you to create custom resources by invoking shell scripts, alleviating the need to create one-off Terraform providers for specific tasks. Many people find it useful for patching gaps in existing providers or creating specific utility functions. In this section we are going to cover how to install the provider and go through some examples of what can be done with it.

### A.1 Installing the Provider

To install a custom Terraform provider you first have to declare that you want to use a custom Terraform provider. Each Terraform module must declare which providers it requires, and this information is something that I normally put in `versions.tf`, as provider requirements are declared in the `required_providers` block of Terraform settings (see Snippet A.1). This will be used to source the provider from the provider registry.

---

<sup>1</sup> <https://github.com/scottwinkler/terraform-provider-shell>

**NOTE** Terraform first checks the local directory and `~/.terraform.d/plugins`, before it checks the provider registry.

### Snippet A.1 provider

```
terraform {
  required_providers {
    shell = {
      source = "scottwinkler/shell"
      version = "~> 1.0"
    }
  }
}
```

We can now install the third-party provider by running a normal `terraform init`.

```
$ terraform init

Initializing the backend...

Initializing provider plugins...
- Finding scottwinkler/shell versions matching "~> 1.0"...
- Installing scottwinkler/shell v1.7.3...
- Installed scottwinkler/shell v1.7.3 (self-signed, key ID 2CAB13AD54B7DF3D)

Partner and community providers are signed by their developers.
If you'd like to know more about provider signing, you can read about it here:
https://www.terraform.io/docs/plugins/signing.html

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
```

## A.2 Using the Provider

Once you have the Shell provider installed, you have access to two new resources, a `shell_script` resource and a `shell_script` data source. These two resources allow you to create custom resources in Terraform by specifying commands that will be run during Terraform CRUD operations. You can also set computed attributes, and reference them from Terraform. For example, a simple data source that can read the current logged in user with `whoami` is:

### Listing A.1 shell script data source

```
terraform {
  required_providers {
    shell = {
      source = "scottwinkler/shell"
```

```

        version = "~> 1.0"
    }
}
}

data "shell_script" "user" {
  lifecycle_commands {
    read = <<-EOF
      echo "{\"user\": \"\$whoami\"}" #A
    EOF
  }
}

output "user" {
  value = data.shell_script.user.output["user"] #B
}

```

#A set output of custom data source

#B reference output here

If you were to run this, you would get:

```

$ terraform apply -auto-approve
data.shell_script.user: Refreshing state...

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

Outputs:

user = swinkler

```

**TIP** this pattern could also be used to read generic environment variables into Terraform variables

Now, I know what you might be thinking: a data source that calls external scripts is not particularly useful or interesting, the same thing could also be done with external data sources or the Null provider. What makes the Shell provider unique is its support for managed resources that implement the full lifecycle of Terraform resources. An example implementation that gets the current weather in London and saves it to a local file is shown in Listing A.2.

### Listing A.2 shell script resource

```

terraform {
  required_providers {
    shell = {
      source = "scottwinkler/shell"
      version = "~> 1.0"
    }
  }
}

resource "shell_script" "weather" {
  lifecycle_commands {
    create = <<-EOF

```

```

    echo "{\"London\": \"$(curl wttr.in/London?format=\"%l:+%c+%t\")\"}\" > state.json
    cat state.json
EOF
delete = "rm state.json"
}
}

output "weather" {
  value = shell_script.weather.output["London"]
}

```

Applying this would query the weather from wttr.in and save it into a local file called state.json:

```

$ terraform apply -auto-approve
shell_script.weather: Creating...
shell_script.weather: Creation complete after 0s [id=bpcrf2dgrkri1bd7rgsg]

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

Outputs:

weather = London: 🌡 +14°C

```

Since it's a normal Terraform resource, it participates in the resource lifecycle by saving state to the state file:

```

$ terraform state show shell_script.weather
# shell_script.weather:
resource "shell_script" "weather" {
  dirty           = false
  id              = "btdk3gdgrkru9f4634h0"
  output          = {
    "London" = "London: 🌡 +14°C"
  }
  working_directory = "."

  lifecycle_commands {
    create = <<~EOT
      echo "{\"London\": \"$(curl wttr.in/London?format=\"%l:+%c+%t\")\"}\" > state.json
      cat state.json
    EOT
    delete = "rm state.json"
  }
}

```

Additionally, you can see that a new file has been created, state.json. This file stores the output of the command, and represents a managed resource.

```

$ cat state.json
{"London": "London: 🌡 +14°C"}

```

A terraform destroy ensures that the state.json file is deleted.

```

$ terraform destroy -force
shell_script.weather: Refreshing state... [id=bpcrg45grkri1sm1kf00]

```

```
shell_script.weather: Destroying... [id=bpcri45grkri1sm1kf00]
shell_script.weather: Destruction complete after 0s

Destroy complete! Resources: 1 destroyed.
```

You can verify that it has been deleted by cat-ing it out once more.

```
$ cat state.json
cat: state.json: No such file or directory
```

### A.3 Final Thoughts

The Shell provider can be used for much more than what we have seen. It supports full CRUD resource lifecycle management and allows you to do almost anything that would normally only be possible by writing a custom Terraform provider. Because it stores stateful information like any other Terraform resource, and supports read and update capabilities, its more versatile than Null resources with attached local-exec provisioners. To give you an idea of what is possible, an example of using the Shell provider for creating a Github repository is shown in Snippet A.2.

#### **Listing A.2 shell script Github repository**

```
variable "oauth_token" {
  type = string
}

provider "shell" {
  sensitive_environment = {
    OAUTH_TOKEN = var.oauth_token
  }
}

resource "shell_script" "github_repository" {
  lifecycle_commands {
    create = file("${path.module}/scripts/create.sh")
    read   = file("${path.module}/scripts/read.sh")
    update = file("${path.module}/scripts/update.sh")
    delete = file("${path.module}/scripts/delete.sh")
  }

  environment = {
    NAME      = "My-Github-Repo-Name"
    DESCRIPTION = "some description"
  }
}
```

**NOTE** for the complete example, refer to the Shell provider documentation<sup>2</sup>

---

<sup>2</sup> <https://github.com/scottwinkler/terraform-provider-shell/tree/master/examples/github-repo>