TERRAFORM

from (beginner to master)

{

// with examples in AWS

}

Kevin Holditch

Terraform: From Beginner To Master: With Examples In AWS

Kevin Holditch

This book is for sale at http://leanpub.com/terraform-from-beginner-to-master

This version was published on 2020-09-30

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

Preface **1**
Thanks 1
Preface 1

Chapter 1 - Introduction to Terraform **2**
What is Terraform? 2
Issues with configuring infrastructure manually 2
Terraform to the rescue 3
Why not just use CloudFormation? 5
What about Chef and Puppet, don’t they solve this problem? 6

Chapter 2 - Installation **7**
Installation 7
Setting up your free AWS Account 8
Setup an AWS user for use with Terraform 8
Setup an AWS Credentials file 9
Install JetBrains IntelliJ Community Edition 9

Chapter 3 - Your First Terraform Project **10**
Code samples 10
Setting up your first project 10
Creating your first infrastructure with Terraform 11

Chapter 4 - Resources **13**
Resources in detail 13
Interpolation syntax 15

Chapter 5 - Providers **18**
Providers in detail 18
Provider best practices 19
More than one instance of the same provider 20

Chapter 6 - Data Sources **22**
Data sources in detail 22
How are data sources useful? 23

Chapter 7 - Outputs **25**
Outputs explained 25
Outputting resource properties 26
Exporting all attributes 27

Chapter 8 - Locals **28**
Locals in detail 28
Locals referencing resources 29

Chapter 9 - Templates and Files **30**
Files 30
Templatefile function 31
Loops in a template 32
Variables 33
Our first Variable 33
Variable defaults 35
Setting a variable on the command line 36
Setting a variable using an environment variable 36
Setting a variable using a file 37
More complex variables 37
Type constraints - Simple Types 38
Type constraints - List 40
Type constraints - Set 41
Type constraints - Tuple 42
Type constraints - Map 42
Type constraints - Object 43
Type constraints - Any type 45

Chapter 11 - Project Layout **46**
Layout 46

Chapter 12 - Modules **48**
Modules 48
Modules in action 48
Returning a complex type from a module 52
Modules using a sub module 53
Remote modules 57

Chapter 13 - Plans **60**
Plans 60
Plan command 65
Auto apply 66

Chapter 14 - State **68**

State 68
Manipulating State 69
Moving a resource from one project to another 70
Remote state 73

Chapter 15 - Workspaces **76**
Workspaces 76

Chapter 16 - Provisioners **79**
Provisioners 79
Null Resources 85

Preface

Thanks

Thank you for choosing to purchase this book through Leanpub. The book is still in build so should you have any constructive feedback about the book so far, how it could be improved, future chapter ideas or you just want to say hi then please email me:

kevin.holditch@gmail.com

Once again many thanks for supporting this project.

Preface

This book will guide you from Terraform beginner to master.

In the modern world where software systems and infrastructure are more complex than ever before, we need a way to manage and configure all of our infrastructure and components that it comprises of. We want to be able to define our infrastructure as code so that it can be checked into source control, shared between team members and used as a source of truth as to how the system looks. We want a tool to manage our infrastructure for us so that we can rid ourselves of manual configuration which is time consuming and error prone.

Terraform is an industry leading infrastructure as code tool that allows you to define your system in code and then run it to make what your infrastructure look exactly how you have defined. Terraform allows you to configure pretty much anything from any cloud including Azure, AWS, GCP to any other component or service like Postgres, Kong or DNSimple to name but a few. The list really is almost endless.

In this book we will work from the ground up, starting with what Terraform is and the problem it solves and then guiding you through each feature of Terraform. In each chapter we will build on your knowledge so you will be able to write a complete project to configure pretty much any infrastructure using any vendor or migrate your existing infrastructure into Terraform.

We will be working with AWS in the book as it is a widely used cloud and makes for great examples. In no way does this mean that after reading this book you will only be able to use Terraform with AWS, quite the contrary. You will be learning Terraform not AWS, with your new found Terraform skills you will be able to successfully use Terraform to configure and setup infrastructure and components from any vendor.

You will truly be able to consider yourself a Terraform master!

Chapter 1 - Introduction to Terraform

What is Terraform?

Terraform is a tool for building, changing, and versioning infrastructure safely and efficiently. It takes your infrastructure you have defined in code and makes it real! The beauty of what Terraform does is that it does not ask you how to get from the infrastructure you have to the infrastructure you want, it just asks you what you want the world to look like and then it does the hard work.

If you are not very familiar with writing code then do not fret. The code that you have to write to configure Terraform is quite different from normal imperative code in languages such as Java or C#. So you can forget about classes and interfaces etc. Instead you can think of the code more as small blocks that represent something in the real world and then a set of properties for that resource to configure it. For example you create a block that represents an AWS EC2 instance and then you can set the properties to say the type of instance you want and the AMI image you want. You do not need to tell Terraform how to do what you want or how to get from what you have now to what you want as you would in an imperative language. All you have to do is tell Terraform what you want and that's it!

Let's walk through a small example:

In your Terraform project you have defined that you want 4 AWS EC2 instances. If you currently have no EC2 instances then when you run Terraform then it will create 4 AWS EC2 instances for you. If you have 3 EC2 instances when you run Terraform then Terraform will only create 1 additional instance and leave the 3 you already had. If you have 5 AWS EC2 Instances, Terraform will delete one. At no point does Terraform ask you how many instances you currently had, Terraform figured it out and then created a plan on how to get from what you have to what you want and then made it happen. This may seem trivial in this example but think about the power that gives you when you extrapolate that out over a whole environment.

Now that we know what Terraform is, lets discuss some of the common problems that occur when you manage your infrastructure by hand and don't use Terraform.

Issues with configuring infrastructure manually

How many times have you worked at a company where every environment (Dev, QAT, Staging, Production etc) has its own personality. You try and test a feature on QAT and you hear "oh that never works on QAT we will have to check that on staging" or "Production is the only environment with a load balancer so that's why we never spotted the bug before". When humans are responsible for keeping environments in sync, things fall between the cracks and the environments drift apart. It is also a lot of manual work to constantly apply changes to each environment. Having environments with different infrastructure causes a number of issues such as you only find bugs on a certain environment and make development hard as you are never testing against production like infrastructure.

Configuring infrastructure manually is very error prone. If you want to try out a new infrastructure configuration then you have to make the change to an environment by hand. If the change is what you want then you have to remember what steps are involved to make the change and then manually apply them to all of your other environments. If you do not like the change then you have to remember how to roll the environment back to how it was. As the process is manual, often the changes are not made exactly the same to each environment which is one of the reasons that environments end up differing and have their own personalities.

It is very time consuming to make the changes manually. If you have several environments and the change is quite complex it can take days to roll that change to each environment.

Once you have an environment when you come to no longer need it destroying it can be very painful. For starters you have to destroy the infrastructure in the correct order as often you cannot destroy a piece of infrastructure if another piece depends on it. You end up becoming a human dependency tree calculator. After a lot of pain you finally think that you have finished destroying the environment only to get a bill from your cloud provider the following month for that piece of infrastructure that you accidentally left running.

Terraform to the rescue

Terraform solves all of these problems because your infrastructure is defined in code. The code represents the state of your infrastructure. When your run Terraform against your code it will update your environment to be exactly how you have specified it in code. Reproducible every time. The machine prospers where humans fail. All of your environments are identical!

Terraform can make all of the changes to your environment very quickly. No longer do you have to wait for days whilst someone follows the run book by hand. A change is made to the code, merged and then instantly can get Terraform to update every environment simultaneously to include the new change.

As your infrastructure is now defined in code you can check it into source control. This means that you can make a change to your code, roll it into an environment using Terraform and try it out. If the change is no good then you can simply go back to the previous version of the code in source control and run Terraform again. Then Terraform handles putting the environment back to how it was. If the change is good then you can check that into source control and roll it into all of your other environments.

Having your infrastructure in code has another major benefit. You can now easily create multiple instances of the same configuration (multiple environments). All of the instances can be created quickly and all will be identical. Being able to create multiple identical environments is a big competitive advantage as it means that each team can have their own environment, you could even have one per person if you wanted! You know that the environment you are testing your software on is exactly the same as production. So there are no sudden surprises due to environment drift!

Terraform is actually split into two parts. One part is the Terraform engine that knows how to get from the state your infrastructure is currently in to how you want your infrastructure to be. The other part is the provider which is the part that talks to the infrastructure to find out the current state and make the changes using the infrastructure's API. Due to the clever way Terraform is split there are providers available for just about everything you can think of. Meaning you can use Terraform to configure infrastructure in AWS, Azure, GCP, Oracle Cloud Platform and just about any other cloud you can think of. It can also be used to configure a huge variety of other components that make up your environment such as Kong, Postgres, Runscope, Auth0, Couchbase, TeamCity you name it there is probably a provider for it. Plus if there is not a provider for it then the really cool part is that you can write your own and then use that in your project.

This means that in a single project you can configure multiple components and infrastructure that sits in multiple clouds. All using the same language (HCL) and all in the same project sitting together. This is such a powerful concept that you can define every aspect of your environment all in the same project and Terraform can work out the order to run and configure each component for you so you do not have to worry about that.

Terraform uses a language called Hashicorp Markup Language or HCL as it known. HCL as you will see is a very simple, easy to read syntax that is completely understandable even to someone looking at it for the first time. This makes it straight forward to read through the code that defines the environment and work out what it is going to do.

Terraform has a massive online community. Having a big online community means that help is never far away. Chances are if you have a problem then you will be able to find the solution from the community. The community also contribute to the providers which is a big reason as to why there is a provider for almost every service you can think of. As the providers are often open source you can raise issues you find on the provider repository and get answers on any issues from the provider author themselves. Often bugs and issues get fixed quickly. You can even fix the provider yourself and run a local fixed build if you need the provider fixed straight away. Due to the way providers are built and run they are normally very quick to encompass new changes to an infrastructure API.

Terraform allows you to see what it is going to do (plan) and await your confirmation before it actually makes any changes. This is a great safety net in case you made a change that you did not mean. It gives you an insight into how Terraform will update your environment to match your desired state (we will cover Terraform plans in much more detail in the book).

With Terraform you can destroy a whole environment and be guaranteed that you are left with nothing. Meaning no more unwanted bill for that piece of infrastructure you forgot to delete. Terraform can calculate the dependency order that infrastructure needs to be deleted in so that it can delete it in the correct order. All automatically and very quickly.

Terraform has a solution if you already have infrastructure and you want to start using Terraform to manage it. You can do this by importing your infrastructure into Terraform. This is great as it allows you to move your infrastructure from being manually setup to being defined in code.

Why not just use CloudFormation?

As this book is going to use AWS for examples I thought it would be prudent to address the question: *Why use Terraform over CloudFormation?* As CloudFormation is an infrastructure as code tool that is doing the same job and it is written by Amazon themselves so surely it is better? Well not exactly. There are a number of reasons why Terraform is a much better choice than CloudFormation for your project.

Terraform is open source and generally moves faster than CloudFormation. Even though CloudFormation is produced by Amazon it can still take a while for a new AWS feature to appear in CloudFormation believe it or not! Whereas the community are amazing at keeping Terraform up to date. This is aided by the fact that each Terraform provider (think of that as a plugin to manage a certain vendor or component) is a separate binary that gets deployed at its own speed (we will cover providers in detail later in the book).

CloudFormation uses JSON or YAML for configuration. Both of these formats are flawed in my view for different reasons. JSON can be quite tricky to read when you have a big object and fiddly to get right due to all of the curly braces. JSON does not allow comments either which means if you want to put a note on something to explain it then you cannot do that. YAML does allow comments and is a bit less verbose than JSON. The big downside of YAML (and anyone that has used it will contest to this) is that YAML is very very fussy about correct indentation. It can have you pulling your hair out trying to get right. If you want to remove a block in the middle of your YAML file it is a nightmare trying to get the indentation correct again. YAML is also hard to follow when you have a large file. It is hard to read it as a human quickly and work out what is going on.

Terraform uses HCL, which has a clean concise syntax. It is very easy to read, allows comments (both inline and block) and is not fussy about spacing, newlines or indentation. That is not to say you cannot use a formatter or an IDE to get it looking neat, it is just that it is not a syntax error if you add an extra space as it can be with YAML. Using HCL you can easily split your project up into multiple files as you see fit. To make the code easier to read and understand when coming to the project.

The killer feature that makes Terraform the obvious choice over CloudFormation is that you can use Terraform to configure all of your infrastructure whereas you can only use CloudFormation for AWS. This means that you can have one tool and project to manage all of your infrastructure. Even if your infrastructure is made up of several components and split across multiple clouds. You can even write your own Terraform provider if you want to configure something that is not currently supported by Terraform. Meaning that you can use Terraform to configure absolutely everything. If you are using CloudFormation then if you want to configure anything other than AWS then you have to use a different tool for that.

What about Chef and Puppet, don’t they solve this problem?

Chef and Puppet are configuration management tools. They are designed to configure and manage software that is running on a machine (infrastructure) that already exists. Whereas Terraform sits at the abstraction layer above that and is designed to setup all of the infrastructure that make up your system such as load balancers, servers, DNS records etc.

As a small aside it is possible to configure software already running on a machine through Terraform using provisioners but this should be used with caution and it is best to leave this type of configuration to specialised tools like Puppet and Chef. *Provisioners and their use case will be covered later in the book.*

Chapter 2 - Installation

In this chapter we are going to walk through installing Terraform on your machine. Then we are going to setup an AWS account and configure Terraform to use it.

Installation

Visit the official Terraform download page and download the latest version for your target platform. Unzip the download to extract the Terraform binary. Terraform runs as a single binary so all you need to do is move the binary so that it is in a folder that is in your path. The follow varies slightly by platform:

Mac OS/Linux

1. Open up a Terminal
2. Change into the downloads directory, normally by running `cd ~/Downloads`
3. Move the Terraform binary into `/usr/local/bin` by running `mv ~/Downloads/terraform /usr/local/bin/`
4. Test the installation by running `terraform version`, if installation is successful then you should see such as `Terraform v0.12.7`

Windows

1. Move the unzipped Terraform binary into your desired folder such as `c:\Terraform`
2. Search for `View advanced system settings`
3. In then window that appears click `environment variables`
4. In the system variables section at the bottom find the path variable, left click it to select it and then click edit
5. In the edit system variable window scroll to the end of the variable value box, ensure that it ends in a `;` then enter the path where you moved the Terraform binary into e.g. `c:\Terraform`

1. Click ok to close all of the windows you have opened

1. Open up a Command prompt by pressing the windows key, typing `cmd` and pressing enter.
2. Test the installation by running `terraform version`, if installation is successful then you should see such as `Terraform v0.12.7`

Setting up your free AWS Account

Due to the fact Amazon change these pages quite a bit, I'm just going to talk through the general process of what you need to do.

1. Head over to https://aws.amazon.com
2. Click on the create Free Tier Account link
3. Fill in your details
4. You will need to register a payment card. This is so that if you go over your free tier Amazon charge you. Do not worry about this if you follow the examples in this book nothing should cost any money. Just remember to delete the infrastructure once you have finished with it. Luckily Terraform can do this for you!
5. I recommend that you turn on 2FA for your newly created AWS log in

Setup an AWS user for use with Terraform

We now need to create an AWS user that we can use with Terraform. For the purposes of this book we are going to create an account which has administrator permissions. This is not recommended for a production setup. I cover best practices for AWS configuration later in the book.

1. Log into your AWS account and you have access and go to the IAM section, you can do this by searching for `IAM` in the search box on the main AWS page and then clicking on the link
2. Select `Users` from the left hand menu
3. Select `Add User` at the top
4. Type in any username you like
5. For access type select `Programmatic access` only
6. Click `Next`
7. On the set permissions screen select ‘

Attach existing policies directly‘

1. Tick `AdministratorAccess` which should be the top of the list
2. Click `Next`
3. Click `Next` again, now you should see a summary of the user you are about to create
4. Click the `Create User` button and the user should be created
5. Store the Access Key Id and Secret Access Key somewhere safe as this is the only time you will see them

Setup an AWS Credentials file

The last thing we need to do is create an AWS Credentials file. This is so that Terraform can get the programmatic credentials for the AWS user we created above.

You need to create a file and with the following text, replacing the two placeholders with the access key id and secret access key you got from AWS when you created your admin user.

```
[default]
aws_access_key_id = <access_key_id_here>
aws_secret_access_key = <secret_access_key_here>
```

Lastly save the file to the path given in the table below based on your OS:

OS	Credentials file path
Windows	`%UserProfile%/.aws/credentials`
Mac OS/Linux	`~/.aws/credentials`

Install JetBrains IntelliJ Community Edition

This last step is completely optional but I would highly recommend it. JetBrains have an awesome IDE called IntelliJ and whats more they provide a free community edition. The great thing about using the IntelliJ IDE is that you can install a plugin that gives you code completion, refactoring and navigation for Terraform files (`.tf` files). This will make your life much easier when you are editing Terraform code.

To setup IntelliJ Community Edition for Terraform:

1. Navigate to the JetBrains IntelliJ download page: https://www.jetbrains.com/idea/download/
2. Click on the `Community Edition` download button
3. Install it by running the download
4. Run IntelliJ and click on the `IntelliJ IDEA` Menu, select `Preferences`
5. On the `Preferences` menu go to `Plugins`
6. In the `Plugins` search box type `HCL`, there should be a plugin for `Hashicorp Markup Language Support`, click `Install` then click `Apply`
7. IntelliJ will now be configured to understand Terraform files

If you do not wish to use IntelliJ, that is fine and you will still be able to follow the examples.

Chapter 3 - Your First Terraform Project

In this chapter we are going to create your first Terraform project. We are not going to cover everything in great detail as we will circle back and cover it in more detail in later chapters. I want the focus of this chapter to be about getting a feel for running Terraform and actually creating some infrastructure with it.

Code samples

All of the code samples given in this book will be on github in the repository: https://github.com/kevholditch/terrafor beginner-to-master-examples.

You can either type them in yourself from the book or feel free to clone the repo or just copy and paste them from github. The examples are grouped into folders at the top level of the repository. I will give the folder name of each sample as it is used.

Setting up your first project

1. Create a folder named `MyFirstTerraformProject` on your hard drive
2. Inside the folder create a file named `main.tf`
3. Inside the file `main.tf` paste the following text (replace <yourname> with your name or something unique):

```
provider "aws" {
  region = "eu-west-1"
}

resource "aws_s3_bucket" "first_bucket" {
  bucket = "<yourname>-first-bucket"
}
```

If you want to copy the code from the sample repository it is in the folder named `first_terraform_project`

That's all we need for our first Terraform project. The Terraform code we have just written will create an S3 bucket in AWS with the name `<yourname>-first-bucket` in the region `eu-west-1`. I reckon you could pretty much guess what it is going to even if you did not know Terraform. That is one of the strong parts of Terraform in that the code is very readable and normally quite obvious what is going to happen.

Lets take a second to explain each part of the code in a bit more detail. In the first 3 lines we are defining the provider that we want to use. Terraform itself is just an engine that knows how to run a provider that conforms to an interface. The Terraform engine is smart and knows how to create dependency trees and plans and it uses the provider to interface with the outside world. As in this book we are going to be using Terraform with AWS we need to configure the AWS provider.

To configure the provider we use the keyword `provider` then follow it with the name of the provider in quotes in this case `"aws"`. We start the provider block by opening a curly brace `{`. We can now specify any parameters we want to configure the provider. To pass a parameter you simply put the name of the parameter followed by an equals sign then the value you want to give the parameter in quotes, in our example we are setting the region this provider will use to be `eu-west-1`. This is the region where the AWS Terraform provider will create all of the infrastructure we define. We then end the provider block with a closing curly brace `}`.

The next block we have defined is a resource. A resource in Terraform represents a thing in the real world. In this case an S3 bucket. To define a resource you start a resource block by using the keyword `resource`. You then follow it with the resource you want to create in quotes. We want to create an S3 bucket so we are using the S3 resource `"aws_s3_bucket"`. If you are following along in IntelliJ and typing in the code you might have noticed that IntelliJ gave you a full list of possible resources once you started typing. You can see the full list on the AWS provider page if you are interested: https://www.terraform.io/docs/providers/aws/index.html. After we have specified the type of resource we want to create we then put another space and then the identifier you want to give that resource in quotes, in our example `"first_bucket"`. We then open the block in the same way that we did for the provider block with an opening curly brace `{`. Next we can give any parameters the resource takes values. We are only setting the name of the bucket. You then end the resource block with a closing `}`.

Creating your first infrastructure with Terraform

The first thing you have to do with a new Terraform project is initialise Terraform for that project. To do this go to your Terminal and cd into the folder where your project is, if you followed this guide exactly then cd into the folder named `MyFirstTerraformProject`. Now run the following command:

```
terraform init
```

You will see some output on the screen as Terraform initialises, then you should see the message `Terraform has been successfully initialized!`. Once you have initialised Terraform you are now ready to create the infrastructure by running:

```
terraform apply
```

After you run the `apply` you will see quite a lot of output from Terraform. You will notice that the apply has paused and is awaiting a response from you.

Lets pause for a second and look at what is happening here. By default when you run `terraform apply` Terraform will look at the code you have written and then compare it to the infrastructure that you currently have (in this case in AWS). Once Terraform has done this it calculates a plan. The plan is what Terraform is going to do to get the real infrastructure from where it is now to how you have specified you want it to be in code. From looking at the plan we can see Terraform is saying if you do this it will create an S3 bucket. You have told Terraform you want an S3 bucket and Terraform went to AWS to check and realised that there is not an S3 bucket in AWS with that name, so it knows that the plan it needs to do is create the bucket. *Plans will be discussed in much more detail later in this book.*

The great thing about this plan is that Terraform presents it to us and then pauses and lets us decide whether we want to go ahead. You can imagine how useful this is if you accidentally make a change that is going to destroy your database! To get Terraform to make these changes and create the S3 bucket type `yes` and press enter.

Once the apply has finished you should see the message `Apply complete! Resources: 1 added, 0 changed, 0 destroyed.`. This is Terraform telling you that it successfully created the S3 bucket for you. Log onto the aws console (website), go to the S3 section and you will see the bucket that Terraform created. Delete the bucket from the AWS console. Now go back to the terminal and run `terraform apply` again. You will notice that Terraform has worked out the S3 bucket is not there anymore so it needs to create it again. At no point did you tell Terraform the bucket was gone, Terraform worked it out. Confirm the apply (by typing `yes`) so the S3 bucket exists again. Now run `terraform apply` again when the bucket is there. You will see Terraform output `Apply complete! Resources: 0 added, 0 changed, 0 destroyed.`. Terraform realises that the state of the world is exactly how you want it to be, so Terraform is saying "nothing to do here!".

To finish up lets destroy the infrastructure we created, don't worry Terraform can take care of that for us. Simply run the command `terraform destroy`. Terraform will present a plan to you of what it is going to destroy and then pause so you can confirm. Type `yes` and press enter. When the destroy finishes you will see a message `Destroy complete! Resources: 1 destroyed.`. This is telling you Terraform has successfully destroyed everything. Log into the AWS console and go to S3 and you will see that the bucket is now gone.

That concludes our first experience with Terraform. I hope that you can start to see the power that Terraform provides and how simple it is to use. Feel free to play around with this project and try changing the properties like the name of the S3 bucket and see what happens. That is a great way to learn. Just remember to run `terraform destroy` when you are finished to ensure that you are not left with any infrastructure running in AWS.

Do not worry if you had more questions about any of the steps we have just been through. We are going to cover everything in much more detail. I wanted to give you a taster for Terraform in action so you could see how powerful it is!

Chapter 4 - Resources

Resources in detail

Resources in Terraform represent a thing in the real world. For example a resource could be an AWS Load Balancer, an alarm in PagerDuty, a policy in Vault, the list is pretty much endless. The resource is the bedrock of Terraform. It allows you to define how you want to create something in the real world. Remember you can create resources that represent things from multiple vendors (for example multiple clouds) in a single project.

Lets take a look in a bit more detail at the resource we defined in the previous chapter:

```
resource "aws_s3_bucket" "first_bucket" {
  bucket = "<yourname>-first-bucket"
}
```

The resource type `aws_s3_bucket` starts with the name of the provider followed by an underscore (`aws_`). This allows you to tell just from the first word of the resource which vendor or component this resource will be created in. Lets take a look at a few other examples:

```
resource "google_folder" "department" {
  display_name = "Department"
  parent     = "organisation/1234567"
}
```

The `department` resource above will create a folder in Google Cloud (GCP). You can see that it starts with `google_` which is the name of the Google Cloud (GCP) provider. Every resource for this provider will start with `google_`.

```
resource "postgresql_role" "my_role" {
  name     = "my_role"
  login    = true
  password = "password123"
}
```

The `my_role` resource above will create a login on a Postgres database, with log in name `my_role` and password `password123`. The resource name starts `postgresql_` as will every resource for the Postgres provider.

If we look back at our S3 bucket resource the last word on the line in quotes was `"first_bucket"`. This is the identifier for that S3 bucket within your Terraform project. The identifier is what we use inside our project to refer to an instance of a resource. You can create multiple instances of the same resource for example you could create many S3 buckets. The identifier gives you a way to reference each one.

The key name value pairs that make up the body of the resource are the properties for the resource. Some properties on the resource are mandatory and some are optional. For an AWS S3 bucket the only mandatory property is the name of the bucket. We could have set more properties on the bucket for example:

```
resource "aws_s3_bucket" "first_bucket" {
  bucket = "kevholditch-first-bucket"
  acl    = "private"

  versioning {
    enabled    = true
    mfa_delete = false
  }
}
```

In the above example (which you can find in the repository folder `resources_example_01` if you want to copy the code across) we are setting the acl to `private` which is basically saying that this bucket will only allow private access. We are also setting two properties for versioning, one to say we are enabling versioning and another to say that you do not require MFA to delete an item on this bucket. With the properties for versioning you will notice that these are nested in another object. This is a design choice by the resource creator that groups all of the versioning properties together. You may also notice that the two properties in the versioning section are booleans (`true/false`). These do not require quotes around them like strings do as we have used for the other two properties.

You can get a full list of all of the properties that are supported from the Terraform provider documentation page, for an S3 bucket it is: https://www.terraform.io/docs/providers/aws/r/s3_bucket.html. You can find the documentation quite easily on Google.

Lets take a look at another resource type so we can examine the other data types that resources can take in their properties:

```
resource "aws_security_group" "my_security_group" {
  name        = "allow_tls"
  ingress {
    protocol    = "tcp"
    from_port   = 443
    to_port     = 443
    cidr_blocks = ["10.0.0.0/16", "11.0.0.0/16"]
  }
}
```

In the resource above we have the two other types of data resources can take numbers and lists. The port properties (`from_port` and `to_port`) are numbers, these are set by just providing the value with no quotes. `cidr_blocks` is a list type, it takes a list of CIDR blocks to restrict for this security group to. You can see that a list is given in the same way a JSON array of strings is created where you surround it in square braces.

Interpolation syntax

Once a resource is created it returns a bunch of attributes. The attributes a resource returns can be found in the `Attributes Reference` section on the documentation page for any resource. This is amazingly useful as it allows you to use the output from one resource as the argument to another resource.

Consider the following project (which can be found in same repository in the folder `resources_example_02`). If you do not want to copy from the example repository then create a new folder, create a single file in the folder called `main.tf` and place the following code:

```
provider "aws" {
  region = "eu-west-1"
}

resource "aws_vpc" "my_vpc" {
  cidr_block = "10.0.0.0/16"
}

resource "aws_security_group" "my_security_group" {
  vpc_id = aws_vpc.my_vpc.id
  name   = "Example security group"
}

resource "aws_security_group_rule" "tls_in" {
  protocol          = "tcp"
  security_group_id = aws_security_group.my_security_group.id
  from_port         = 443
  to_port           = 443
  type              = "ingress"
  cidr_blocks       = ["0.0.0.0/0"]
}
```

This project creates an AWS VPC with CIDR block `10.0.0.0/16`. Then it defines a security group (`aws_security_group`). In the definition of the security group notice that the value of `vpc_id` is set to `aws_vpc.my_vpc.id`. The value of `aws_vpc.my_vpc.id` is not known before we run the project as AWS will randomly assign it when we create the VPC. By referencing the VPC we created it allows us to use this value even though we do not know what it will be until we run the project.

The format of using an output attribute from a resource is `<resource_type>.<resource_identifier>.<attribute_name>`. In the VPC id example we are getting the output from an `aws_vpc` resource type, with the identifier name `my_vpc` and we want to get the `id` attribute value. So hence we end up with `aws_vpc.my_vpc.id`. It is worth noting here that this syntax was greatly simplified in Terraform version `0.12`. Which is the syntax all of the examples in this book will be using.

Next in our project we define a security group rule (`aws_security_group_rule`) to allow ingress traffic on port `443`. In the `aws_security_group_rule` we need to reference the id of the security group that we want to put this rule in. We can use the same technique as we did when we referenced the id of the VPC. Lets work through how to figure this out together. It will start with the type of the resource we want to reference `aws_security_group`. Next we use the identifier to specify which instance of the security group we want to use which is `my_security_group`. Lastly we use the attribute of that property we want to use, which is `id`. This leads use to build the expression `aws_security_group.my_security_group.id` which we can use for the value of the property `security_group_id` inside the `aws_security_group_rule` resource.

The neat thing about using interpolation syntax to reference the attribute of a resource in another resource is that it allows Terraform to work out the dependency order of the resources. From our HCL above Terraform can determine that first it needs to create the VPC because it needs the id that AWS assigns to the VPC in order to create the security group. It then knows that it needs to create the security group next as it needs the id of the security group in order to create the security group rule. Terraform uses this information to build up a dependency graph and then tries to run in parallel as much as possible.

To illustrate the way Terraform can create a project in parallel consider what happens when we add a new security group rule to our project above (folder `resources_example_03` in the example repository).

```
resource "aws_security_group_rule" "http_in" {
  protocol          = "tcp"
  security_group_id = aws_security_group.my_security_group.id
  from_port         = 80
  to_port           = 80
  type              = "ingress"
  cidr_blocks       = ["0.0.0.0/0"]
}
```

When you run the project now with Terraform, it will realise that it can create both security group rules in parallel. Once the security group they both depend on is created, it will be able to create both of the rules together. This feature of Terraform makes performance very good. It may seem quite obvious in this example but as a project grows it can be quite impressive at how much Terraform can run in parallel.

Chapter 5 - Providers

Providers in detail

A provider in Terraform is a connection that allows Terraform to manage infrastructure using a pre defined interface. This abstraction is very powerful as it means the Terraform engine which understands how to read state from a provider, read HCL code and then work out how to get to the desired state, is completely separate from the provider. It allows anyone to write a provider to connect to anything (as long as there is a programmable way to talk to it). All the provider writer has to do for each resource they want to allow Terraform to control is provide Terraform with a way to create it, read it and delete it. *Note update is actually optional as Terraform can always delete and then create the resource if update is not provided.*

Due to the provider model that Terraform employs, providers are not part of the main Terraform source code. They are separate binaries that live in their own repositories and can move at their own speed. This means that if a provider needs to release a bug fix or new feature they can just release it. They do not need to coordinate a release of the main Terraform code base.

A provider is defined using a provider block. You have already used a provider block in the examples so far in this book, it looks like:

```
provider "aws" {
  region = "eu-west-1"
}
```

The provider block is very simple it starts with the keyword `provider` to indicate that this is a provider block. You then have to give the name of the provider that you are using. In this case we are using the `aws` provider so we put `"aws"`. You then use a `{` to open the provider block. Inside the provider block you can put all of the configuration you want for the provider. For the AWS provider the only property that we are configuring is the region. This will be the region that we are going to create our AWS resources in. You then end the provider block with a closing `}`.

Create a new folder and inside that folder create a file called `main.tf` and place only the above provider block in that file (folder `providers_example_01` in the example repository if you want to copy the code). Go to the command line and navigate to the new project that you created and run `terraform init`. You will notice that Terraform downloads the AWS provider automatically, you will see `Downloading plugin for provider "aws" (hashicorp/aws) 2.27.0...`. You may be wondering how that happens? Hashicorp (the company who make Terraform) host a registry which contains the most popular providers. If the provider that you use is in the registry then all you have to do to use it is define a provider block that sets up that provider and run `terraform init` to download it. You can see a full list of providers that are in the registry on the Hashicorp Terraform site: https://www.terraform.io/docs/providers/index.html.

You may be wondering where Terraform puts this provider? It puts it inside the project where you are currently working in a special folder called `.terraform`. The relative path on my Macbook to the AWS provider that gets downloaded when I run `terraform init` is `.terraform/plugins/darwin_amd64/terraform-provider-aws_v2.27.0_x4`. The platform part (3rd folder in) may vary for you depending on the platform you are running on. The provider is actually a separate binary which Terraform calls out to at run time to do its work. As an interesting aside the name of the provider binary is always in the format `terraform-provider-<NAME>_vX.Y.Z`. Terraform uses this convention to search for providers on your machine, so that it knows if you have a particular version of a provider when you run `terraform init`. It uses this information to know whether or not to download it.

Provider best practices

When you ran `terraform init` you may have noticed the warning message that Terraform printed out where it said that it recommends that you use a version constraint on the provider.

To fix this create a new Terraform project by creating a new folder and creating a single file called `main.tf` and placing the following code inside (or grab the code inside `providers_example_02` from the examples repository):

```
provider "aws" {
  region  = "eu-west-1"
  version = "~> 2.27"
}
```

Go to your terminal and go into the folder you have created and run `terraform init`. You will see that Terraform downloads the AWS provider but this time there will not be a warning given about the version constraint. The version constraint `~> 2.27` means any minor version higher is ok to upgrade to but no major version and no beta versions. It effectively means any version up to but not including `3.0.0` is ok. Every time you run `terraform init` Terraform will go up to the registry and check what the latest version is of all of the providers you have defined. It will then look on your machine and see what version you have installed. If you have no version constraint on the provider then you are saying to Terraform that you always want Terraform to download the latest version. This can be dangerous (as the warning message tells you) as it will mean that Terraform will get the latest version even if it is a major version update which means it will contain breaking changes. Therefore you should always use a version constraint on the provider to ensure that Terraform will not upgrade the provider to a new major version. Minor version updates are ok as they will not contain a breaking change. So in our example if version `2.3` gets released Terraform will upgrade our project to that version of the provider when we next run `terraform init`.

The general advice is to use `~> X.YY` notation for your provider. Which will give the semantics described above, which is generally what you want. If you are unsure of what to specify then leave off the version parameter, run `terraform init`. Once Terraform downloads the provider it will tell you what version it has downloaded in the output. Then put the version of the provider it has downloaded after ~> in the version constraint.

Terraform is quite flexible on what you can do with the version constraint as the below list shows:

- `>= 1.2.0`: version 1.2.0 or newer
- `<= 1.2.0`: version 1.2.0 or older
- `~> 1.2.0`: any non-beta version >= 1.2.0 and < 1.3.0, e.g. 1.2.X
- `~> 1.2`: any non-beta version >= 1.2.0 and < 2.0.0, e.g. 1.X.Y
- `>= 1.0.0`, <= 2.0.0: any version between 1.0.0 and 2.0.0 inclusive

More than one instance of the same provider

On the AWS provider as `region` is a required parameter you may be wondering how you create resources in different regions and if that is possible in a single Terraform project? Well yes it is. To do this you simply create multiple instances of the same provider.

Create a Terraform project on your machine by creating a new folder and creating a single file in that folder called `main.tf` (or grab the project from the `providers_example_03` folder in the examples repository), then copy in the following code:

```
provider "aws" {
  region  = "eu-west-1"
  version = "~> 2.27"
}

provider "aws" {
  region  = "eu-west-2"
  alias   = "london"
  version = "~> 2.27"
}

resource "aws_vpc" "ireland_vpc" {
  cidr_block = "10.0.0.0/16"
}

resource "aws_vpc" "london_vpc" {
  cidr_block = "10.1.0.0/16"
  provider   = "aws.london"
}
```

As you can see we are defining two instances of the AWS provider. The once pointing at the region `eu-west-1` and once at the region `eu-west-2`. For the one that is pointing at `eu-west-2` you will notice that we are setting the `alias` property to `london`. Alias is a property you can set on any provider block, it is no way special to the aws provider. What this gives us is a way to distinguish between the two providers. Once you define two or more instances of the same provider every definition after the first must have an alias set.

After we have defined the two AWS providers we are creating a VPC called `ireland_vpc`, with the CIDR block `10.0.0.0/16`. As we have not told Terraform which provider instance to use for this resource Terraform will pick the instance of the AWS provider where you have not defined an alias. Which means that this VPC will be created in the region `eu-west-1` (Ireland). The second VPC we have defined with the identifier `london_vpc` has a CIDR block of `10.1.0.0/16`. This time we have set the `provider` property on the resource to `aws.london`. This means that terraform will use our second AWS provider which points to the region `eu-west-2`, so when we run the project by doing `terraform apply` this VPC will be created in `eu-west-2`.

Every resource has a `provider` property that you can set. The format of the value is set by `<provider_name>.<provider_alias>`. For our example when we created the VPC in london we were using the AWS provider and the alias was london hence we set the provider property to `aws.london`.

You be wondering if there is a way to explicitly tell Terraform to use the default provider even though you do not need to. To do that you can set the `provider` property to the provider name. So for example we could have defined the `ireland_vpc`:

```
resource "aws_vpc" "ireland_vpc" {
  cidr_block = "10.0.0.0/16"
  provider   = "aws"
}
```

This would have given us exactly the same output. Due to this being extra code for no added value no one in the community writes their HCL in this way. Less is more, and it is cleaner and easier to read if you omit the provider property when using the default provider instance.

To run the above project go to a terminal and change into the directory where you have placed the `main.tf` file. Run `terraform init` to initialise the project and download the AWS provider. Next run `terraform apply` then confirm by typing `yes` and pressing enter. This will create the two VPCs, one in `eu-west-1` and one in `eu-west-2`. Log into the AWS Console and navigate to find the VPC section. Switch into the `eu-west-1` and `eu-west-2` regions and you will see the VPCs that Terraform created. Once you are done with the project do not forget to run `terraform destroy` and then confirm with `yes` to delete all of the resources in this project.

Chapter 6 - Data Sources

Data sources in detail

A data source in Terraform is used to fetch data from a resource that is not managed by the current Terraform project, so that value can be used in the current Terraform project. You can sort of think of it as a read only resource that already exists. The object exists but you want to read some properties of that object for use in your project.

Lets dive into an example. Create a new folder on disk, create a file called `main.tf` inside it and paste in the following code (or grab the code from the folder `data_source_example_01` in the samples repository):

```
provider "aws" {
  region  = "eu-west-1"
  version = "~> 2.27"
}

data "aws_s3_bucket" "bucket" {
  bucket = "kevholditch-already-exists"
}

resource "aws_iam_policy" "my_bucket_policy" {
  name = "my-bucket-policy"

  policy = <<POLICY
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "s3:ListBucket"
      ],
      "Effect": "Allow",
      "Resource": [
        "${data.aws_s3_bucket.bucket.arn}"
      ]
    }
  ]
}
POLICY
}
```

As you can see from the above project a data source block starts with the word `data`. The next word is the type of data source. We are using a `aws_s3_bucket` data source, which is used to look up an S3 bucket. After the data source type, we give the data source an identifier in this case `"bucket"`. The identifier is used to reference the data source inside the Terraform project. The data source block is then opened with a `{`. You then specify any properties you want Terraform to use to search for the resource. We are using the complete name of the S3 bucket we are looking for. You then close the data source block with `}`.

Rather than creating the bucket as we did before this time we are referencing a bucket that already exists. So before you run the above project you will need to create an S3 bucket with the name that you specify inside the data block. In the example above the bucket would be called `kevholditch-already-exists`. Name the bucket anything you want but then paste the name into the `bucket` property in the data source.

At the bottom of this project we are creating an AWS IAM policy which gives permissions to list the bucket that we looked up in the data source. There are a couple of new concepts in the `aws_iam_policy` resource that I want to introduce. The IAM policy itself is a multi line string enclosed in between `<<POLICY` and `POLICY`. This is how you define a multi line string in Terraform. You open the multi line string with `<<` then you place any identifier you wish as a single word. I have used `POLICY` in the example above as I am defining an IAM policy but you could have used anything like `<<STATEMENT` or `<<IAM`. You then start your multi line string on the next line and to finish it you use the opening identifier without the `<<`. *Note the closing marker must be at the start of a new line otherwise it is a syntax error.*

Inside the IAM policy we are using the S3 bucket data source. We are taking the arn from the S3 bucket so that we can use it in our IAM policy. You will notice that to get the value we are using the interpolation syntax `${data.aws_s3_bucket.bucket.arn}`. The opening `${` and closing `}` is needed because we are inside a multi line string so it is telling terraform that we want it to evaluate the value of this and not use it as a string literal. The format of a data source expression is `data.<data_type>.<data_identifier>.<attribute_name>`. You can get a full list of the attributes that a data resource provides from the documentation website of the provider.

To run this project go to the terminal and cd into the folder where you created the `main.tf` file. Run `terraform init` to initialise Terraform and then `terraform apply`. When Terraform runs you will see that it only created a single resource, the IAM policy. This is because the S3 bucket we are using is created outside of Terraform.

How are data sources useful?

As your Terraform project gets large it can be sensible to break it up into smaller projects to make it easier to maintain. When this happens you can use data sources to reference resources created in other Terraform projects and still use them. In this case it would always be better to use a data source than to compute the arn yourself which would be possible with something like an S3 bucket. This is because you want Terraform to fail if for some reason the bucket no longer exists. By using a data source you get this behaviour.

Imagine you want to create a new AWS EC2 instance using an AMI image from a private repository. You could hard code the name of the AMI image when creating the instance and then manually update it when a new AMI image is released. This would work but it would be quite cumbersome and would require a code change every time you wanted to use the latest version of the AMI image. By using a data source you could set it up so that it always reads the repository and gets the latest version of the AMI image when you run Terraform. You could then reference that data source when creating the EC2 instance and ensure that you always have the latest version of the image.

Another reason you may want to use a data source is if you are migrating existing infrastructure to Terraform and you want to reference a resource that is not part of your Terraform project yet. As previously stated it is always better to use a data source rather than compute the value yourself. You want Terraform to know that there is a dependency on the resource. As you want your Terraform apply to fail if the resource cannot be found or if the attribute it returns changes then Terraform will realise when you run apply and update your project with the new value.

Chapter 7 - Outputs

Outputs explained

An output in your Terraform project shows a piece of data after Terraform successfully completes. Outputs are useful as they allow you to echo values from the Terraform run to the command line. For example, if you are creating an environment and setting up a bastion jump box as part of that environment then its handy to be able to echo the public IP address of the newly created bastion to the command line. Then after the Terraform apply finishes you get given the IP of the newly created bastion ready for you to ssh straight onto it.

Lets start with an example of outputs. Create a new folder to put our new Terraform project into and create a single file called `main.tf` and paste in the following code (or grab the code from `outputs_example_01` folder inside the examples repository):

```
output "message" {
  value = "Hello World"
}
```

Try running this project by opening your terminal. Changing directory into the folder that you created where the `main.tf` file is and then running `terraform init` and `terraform apply`. You will see that Terraform runs and then prints the following:

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

Outputs:

message = Hello World
```

A couple of interesting things just happened. Firstly, did you notice that Terraform did not pause to ask you if you wanted to do the apply? The reason for this is that Terraform realised there was nothing to do so therefore there was nothing to ask you! You can see from the message above that Terraform states that nothing changed (`0 added, 0 changed, 0 destroyed.`). You then see `Outputs:` and under there Terraform prints out the values of all of the outputs you have defined. We defined a single output with the identifier `message` and gave it the value `Hello world` so that is what Terraform printed.

To define an output you open an `output` block by using the `output` keyword. You then start the output block with `{`. You are only allowed to set a single property called `value`. Whatever value you give to the `value` property will be outputted to the console after a successful Terraform apply. You then close the output block with `}`.

Note outputs are used in modules too and have slightly different semantics, this is covered in the chapter on modules.

Outputting resource properties

The first example is pretty basic and in the real world probably not very useful. Outputs are much more useful when used to output the values of resources that have been created as part of a Terraform run. Lets create another Terraform project and do that. Create a new folder and create a single file called `main.tf` then paste in the following code (or copy the folder `outputs_example_02` in the examples repository):

```
provider "aws" {
  region = "eu-west-1"
}

resource "aws_s3_bucket" "first_bucket" {
  bucket = "kevholditch-bucket-outputs"
}

output "bucket_name" {
  value = aws_s3_bucket.first_bucket.id
}

output "bucket_arn" {
  value = aws_s3_bucket.first_bucket.arn
}

output "bucket_information" {
  value = "bucket name: ${aws_s3_bucket.first_bucket.id}, bucket arn: ${aws_s3_bucke\
t.first_bucket.arn}"
}
```

Lets walk through the above code. The `provider` and `resource` should be familiar to you. We are simply defining the AWS provider to be used with the `eu-west-1` region and setting up an S3 bucket. Feel free to change the name of the bucket to whatever you wish. Next we define an output called `bucket_name`. In the bucket_name we are going to output the name of the bucket by using the attribute of the S3 bucket resource that we create. We use the same technique to output the ARN of the bucket that we create in the output `bucket_arn`. In both of those examples because we are directly using the attribute we can just set it equal to `value` without any quotes. The last output `bucket_information` prints an interpolated string which will give us the bucket name and bucket arn. As this value is a string with interpolated values we have to surround in quotes and `${ }`.

Open a terminal, go into the directory where you have defined that project, run `terraform init`, `terraform apply` and confirm by typing `yes` and pressing enter. Terraform runs, creates the S3 bucket and gives me the following output under the `Outputs:` heading:

```
bucket_arn = arn:aws:s3:::kevholditch-bucket-outputs
bucket_information = bucket name: kevholditch-bucket-outputs, bucket arn: arn:aws:s3\
:::kevholditch-bucket-outputs
bucket_name = kevholditch-bucket-outputs
```

Terraform got the values from the S3 bucket that it created and outputted them when the run completed. Terraform prints the outputs in alphabetical order, not the order that you define them in your project. That is a good point to make, that Terraform does not care which order you define the blocks in your project. Try reordering them and running `terraform apply` again. You will notice that Terraform will say that there is nothing to do.

Exporting all attributes

As of Terraform `0.12>` (which this book is based on), Terraform allows you to output an entire resource or data block. To do this take the example that we just had and add the following output (in the examples repository it is `outputs_example_03` if you want to just get the code):

```
output "all" {
  value = aws_s3_bucket.first_bucket
}
```

Run the project again (`terraform apply`) and you will notice that you see an output called `all` that has all of the attributes that are exported by the `aws_s3_bucket` resource. Sometimes it can be handy just to output the whole resource to the console. Normally when you are debugging something and you want to see what the value of one of the properties is.

Chapter 8 - Locals

Locals in detail

A local is Terraform’s representation of a normal programming languages’s variable. Confusingly Terraform also has a concept called a variable which is really an input, variables are covered in chapter 10. A local can refer to a fixed value such as a string or it can be used to refer to an expression such as two other locals concatenated together or the attribute of a resource that you have created. Lets dive into an example. Create a new folder in your workspace and create a single file inside it called `main.tf` and copy in the following code (or if you want to grab the code from the samples repository copy the folder `local_example_01`):

```
provider "aws" {
  region = "eu-west-1"
}

locals {
  first_part  = "hello"
  second_part = "${local.first_part}-there"
  bucket_name = "${local.second_part}-how-are-you-today"
}

resource "aws_s3_bucket" "bucket" {
  bucket = local.bucket_name
}
```

In the code above we define a local block by using the keyword `locals` and then an opening `{`. We then define each local on a new line by giving it a name. The first local we define is called `first_part`. You then follow it with an `=` and then give it a value. For the `first_part` local we are giving it the value of the string literal `hello`. For the second local `second_part` we are using the value `"${local.first_part}-there"`. As the whole expression is inside quotes we need to use the `${` and `}` around our expression so Terraform evaluates it. To reference a local you use the expression syntax `local.local_identifier`. So the `second_part` local will be set to “hello-there”. In the `bucket_name` local we are using the `second_part` local in the expression `"${local.second_part}-how-are-you-today"` which will evaluate to `hello-there-how-are-you-today`.

At the bottom of the project we are defining an S3 bucket and setting the name to `local.bucket_name`, so this will create an S3 bucket with the name `hello-there-how-are-you-today`. Note we do not need the `${` and `}` as we are not inside quotes here. We could also have set the bucket to

"${local.bucket_name}", which would have evaluated to the same thing. But since Terraform 0.12> we can now omit the ${ and } for a single line expression where we are using the whole value. Which I think makes the code cleaner and easier to read. As we are defining our infrastructure as code the easier it is to read the better.

To run the project go to the terminal and cd into the folder where you created the file main.tf. Run terraform init and then terraform apply. When prompted say yes if you want to run in the project. This will create the S3 bucket with the name hello-there-how-are-you-today. To destroy this infrastructure run terraform destroy and then confirm.

Locals referencing resources

Locals can reference the output of a resource or a data source by using the expression syntax we have learnt. This allows you to give something a more meaningful name or to combine outputs from different resource and data source attributes to build up a more complex value.

Chapter 9 - Templates and Files

Files

We have seen in a previous chapter how you can place a multi line string as a value for a property. This is useful for something like an IAM policy. It can be even cleaner to move the value out into a file and then reference that file from your project. Doing this can remove the clutter from your project and make it much more readable.

Lets see an example of using files, create a new folder, create a file called `main.tf` (or copy the code from the folder `file_example_01` in the examples repository) and paste in the following code:

```
provider "aws" {
  region  = "eu-west-1"
  version = "~> 2.27"
}

resource "aws_iam_policy" "my_bucket_policy" {
  name = "list-buckets-policy"
  policy = file("./policy.iam")
}
```

Next if you are creating the code yourself, create another file called `policy.iam` and paste in:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "s3:ListBucket"
      ],
      "Effect": "Allow",
      "Resource": "*"
    }
  ]
}
```

This IAM policy creates a policy that gives list bucket permission to any bucket. If you look at the Terraform code you will see we are configuring the AWS provider as we are going to be connecting to AWS. Then we are defining an AWS IAM policy. Instead of placing the policy inline as we did in a previous chapter, we are referencing the policy from the file `policy.iam`. To do this we are calling the `file` function and passing in the argument as to where the file is. *Note this is a relative file path from our current location.*

When we run this project by doing `terraform init` and `terraform apply` and confirm by typing `yes`, Terraform will create the IAM policy you can see in the file and name it `list-buckets-policy`.

Templatefile function

Sometimes we want to use a file but we do not know all of the values before we run the project or some of the values are dynamic and generated as a result of a created resource. To be able to use dynamic values in a file we need to use the `templatefile` function.

The `templatefile` function allows you to define placeholders in a template file and then pass their values at runtime.

Lets dive into a simple example to see how it works… Create a new folder and create a file called `main.tf` (or copy the `templatefile_example_01` folder from the examples repository) and paste in the following code:

```
locals {
  rendered = templatefile("./example.tpl", { name = "kevin", number = 7})
}

output "rendered_template" {
  value = local.rendered
}
```

Next if you are creating the code yourself and not using the examples repository then create another file in the directory called `example.tpl` and place the following text:

```
hello there ${name}
there are ${number} things to say
```

Lets walk through the code we have written so we can understand what is going on. We are defining a local (as we learnt in the previous chapter) called `rendered`. We are setting the value of the local (remember a local can have a value that is an expression) to the result of calling the `templatefile` function. The template file function takes two arguments. The first argument is the path to the template file, in this example we are using the relative path between the `main.tf` and the template file `example.tpl` so the path is `./example.tpl`. The next parameter is a set of variables that you want replacing in your template. We are passing in a value for `name` (`kevin`) and `number` (`7`). *Note you could* *set the value of these variables to any expression such as another local or an attribute from a created resource or a data source.*

If you look at the example template code we use the syntax `${<variable_name>}` when we want to reference a passed in variable. Terraform will replace `${name}` in our template with the value passed in for `name`, it will do the same with `${number}`.

We are then using an `output` to output the rendered value of the template out to the terminal. This is an easy way for us to see how the `templatefile` function works.

Lets see templates in action. Go to your terminal, change directory into the new project folder you have just created and run `terraform init` and then `terraform apply`. You will see the following output rendered:

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

Outputs:

rendered_template =
hello there kevin
there are 7 things to say
```

You can see that Terraform replaced the placeholders in our template with the values that we provided to the `templatefile` function. So you see the message with `kevin` and `7` in it rather than the placeholders.

Loops in a template

You can pass in an array of values into a template and loop round them. Lets take a look at an example of how to do that. Create a new folder, create a file called `main.tf` (or grab the code from the folder `templatefile_example_02` in the examples repository) and paste in the following code:

```
output "rendered_template" {
  value = templatefile("./backends.tpl", { port = 8080, ip_addrs = ["10.0.0.1", "10.\
0.0.2"] })
}
```

Next create a file called `backends.tpl` and paste in the following:

```
%{ for addr in ip_addrs ~}
backend ${addr}:${port}
%{ endfor ~}
```

This time we are just rendering the template straight to the output. Notice that we are passing in an array for `ip_addrs`. In the template we are then looping around the `ip_addrs` array by using a `foreach` loop. To do a `foreach` loop you use the syntax `%{ for <var> in <variable_name ~}` where `<var>` is any identifier you want to use to reference the current looped item and `<variable_name>` is the name of the array that is passed into the template. All of the lines you now write are inside the loop until you signal the end of the loop by putting `%{ endfor ~}`. Notice that inside the loop we are using the current value from the `ip_addr` array and we are always referencing the `port`.

When we run the above project (by doing `terraform init` and `terraform apply`) you will notice the following output is rendered for the template:

```
backend 10.0.0.1:8080
backend 10.0.0.2:8080
```

The word `backend` is constant as that is just text so we see that for each iteration around the loop. The current element of the `ip_addr` array is then printed followed by the passed in port, which always has the value `8080`.

The fact that we can combine loops with interpolated values means that you can write some quite elaborate templates. These can be really useful for programmatically generating things like IAM policies, where you can pass in the ARN of resources that Terraform creates and use them as part of the template.

Chapter 10 - Variables

Variables

A Variable in Terraform is something which can be set at runtime. It allows you to vary what Terraform will do by passing in or using a dynamic value.

Our first Variable

Lets dive into an example of how to use variables, so we can learn how they work. Create a new folder and create a file called `main.tf` inside that folder (or simply grab the code from the examples repository in the folder `variables_example_01`) and copy in the following code:

```
provider "aws" {
  region  = "eu-west-1"
  version = "~> 2.27"
}

variable "bucket_name" {
  description = "the name of the bucket you wish to create"
}

resource "aws_s3_bucket" "bucket" {
  bucket = var.bucket_name
}
```

You declare a variable by using the keyword `variable` then you specify the identifier for the variable in quotes, we are using `"bucket_name"` as the identifier. Inside the variable block we are adding a description to describe to the user what this variable is used for. The `description` parameter is completely optional, we could have defined the variable as follows:

```
variable "bucket_name" {}
```

The reason that it is a good idea to provide a description as it gives the user some instruction as to what values to use and what the variable is used for.

To use the value of a variable in your Terraform code you use the syntax `var.<variable_identifier>`. You can see that we are setting the bucket property on the `aws_s3_bucket` resource to the value of our variable `var.bucket_name`. This means that whatever value we give our variable Terraform will use as the name of the S3 bucket.

Run the project by opening your terminal, changing into the directory where you have created the project and running `terraform init` and then `terraform apply`. Terraform will pause and you will see the following output:

```
var.bucket_name
  the name of the bucket you wish to create

  Enter a value:
```

Terraform has paused because it is asking you to provide a value for the variable. The variable name is printed to the screen and underneath is the description we provided. *Note if you do not set a description then only the variable name will be shown here.* Type in a bucket name that you think will be unique. I used `kevholditch-variable-bucket` but it really doesn't matter what value you use. Press enter and then you will notice that Terraform will pause again and ask if you want to apply the changes. Type `yes` and create the bucket. Terraform will have created the bucket with a name of whatever value you gave it.

Once you have run the project go ahead and destroy it again by running `terraform destroy`. Terraform will ask you for a value for the variable again, it doesn't actually matter what value you give it this time as the variable is not needed for Terraform to destroy the bucket. Press enter and then type `yes` when Terraform asks you if you really want to destroy the project.

Variable defaults

Create another project and create a file called `main.tf` (or copy the code from `variables_example_02`) and paste in the following code:

```
provider "aws" {
  region  = "eu-west-1"
  version = "~> 2.27"
}

variable "bucket_name" {
  description = "the name of the bucket you wish to create"
}

variable "bucket_suffix" {
  default = "-abcd"
}

resource "aws_s3_bucket" "bucket" {
  bucket = "${var.bucket_name}${var.bucket_suffix}"
}
```

We have extended the first example and added a second variable `"bucket_suffix"` and we have set its default to `"-abcd"`. Setting a default on a variable means that if you do not provide a value for that variable then the default will be used. We are then using the value of the bucket_name variable concatenated with the value of the bucket_suffix variable for the value of the bucket name. As we are using the values inside a string we need to surround our variables with `${` and `}`. Otherwise Terraform will not use the value of the variable and instead would just print the string `var.bucket_name`.

Run the project (`terraform init`, `terraform apply`). Terraform will ask you to provide a value for `bucket_name` as before. Enter a name for the bucket and press enter. Notice that Terraform will now ask you to confirm the run by typing `yes`. Confirm the run and press enter. Terraform will then go and create the bucket. You may be wondering why Terraform never asked you for a value for `bucket_suffix`, well it is because Terraform does not need a value for `bucket_suffix` as you already gave it a default value of `-abcd`. The end result is that a bucket will be created with whatever name you enter for the bucket name with `-abcd` on the end of it.

Setting a variable on the command line

Lets continue working with the project we have just created (`variables_example_02` in the examples repository). We are now going to learn how we can change the value of `bucket_suffix`. As when you run the project Terraform does not ask you for a value for it as we have just learnt.

The first way we can set a value for `bucket_suffix` is by providing it on the command line. Run the following command `terraform apply -var bucket_suffix=hello`. Terraform will ask you for a value for `bucket_name` as you haven't given it one and it does not have a default. When the project runs Terraform will now create a bucket with whatever name you gave it with `hello` on the end.

To set the value of a variable on the command line you use the `-var` flag followed by the variable name and the value you wish to use. If we want to set both of the variables on the command line then we can do that with the command `terraform apply -var bucket_name=kevholditch -var bucket_suffix=foo`. In this command we are giving a value to both the `bucket_name` and `bucket_suffix`. If you runrun the project with the command to set both variables then Terraform will not stop to ask you a value for the `bucket_name`. This is because you have now provided one. Terraform will stop and ask for a value of any defined variable that does not have a default value or a value set via the command line or one of the other ways we are going to learn.

Setting a variable using an environment variable

Another way you can set the value of a variable is by using an environment variable. To do this set an environment variable in your terminal using the convention `TF_VAR_<variable_identifier>`. So for our project (`variables_example_02` in the examples repository) lets set environment variables for each of the variables. Follow the instructions below based on your OS:

Mac OS/Linux

```
export TF_VAR_bucket_name=kevholditch
export TF_VAR_bucket_suffix=via_env
```

Windows

```
set TF_VAR_bucket_name=kevholditch
set TF_VAR_bucket_suffix=via_env
```

Once you have set the environment variables run `terraform apply`. You will see that now Terraform will not ask you for a value for either variable and will use the values from your environment variables.

Setting a variable using a file

The next way that you can set the value of variables is by using a file. Create a new file in the project called `terraform.tfvars` (or you can copy the project `variables_example_03` from the samples repository). We are going to use the same Terraform code in the `main.tf` we had before.

Inside the `terraform.tfvars` file place the following:

```
bucket_name = "kevholditch"
bucket_suffix = "from_file"
```

`terraform.tfvars` is a special file name that Terraform looks for to discover values for variables. Terraform will look in this file and use any values given for a variable. To set the value of the variable you simply put the variable identifier and then an equals sign and the value you want to give it. We are setting the value of both `bucket_name` and `bucket_suffix` in our file. So now when we run the project, Terraform will use those values for the variables and not ask us for them.

The other way we could have named our file was anything ending in `.auto.tfvars`. Terraform examines files with that ending for variables being set. It is also possible to define multiple files and put the value for different variables in each of them.

More complex variables

Lets look at a more complex example using a map and selecting a value from it with variables. To do this create a new project and create a file called `main.tf` (or copy the folder `variables_example_04` from the examples repository) and paste in the following code:

```
variable "instance_map" {}
variable "environment_type" {}

output "selected_instance" {
  value = var.instance_map[var.environment_type]
}
```

Next create a file called `terraform.tfvars` where we can give these variables values:

```
instance_map = {
  dev = "t3.small"
  test = "t3.medium"
  prod = "t3.large"
}

environment_type = "dev"
```

In our variables file we are setting `instance_map` to a map. A map is a collection of values indexed by a key name. We have set 3 keys in our map `dev`, `test` and `prod`. The values we have given for each of these keys are instance types to use in AWS. This map could be used to set the instance size based on the type of environment you are creating. Next we are setting the `environment_type` variable to `dev`. Look at the Terraform code we have written and you will see that we are defining the two variables `instance_map` and `environment_type`. At the bottom we are outputting the value in the map for the key specified by the `environment_type` variable.

If you run this project as is then it will output `selected_instance = t3.small`. This is because `t3.small` is the value in the map for the key `dev` and we have set the `environment_type` to `dev`. Change the environment type to one of the other values in the map, run the project again and you will see the output change.

As the map of instances is a variable we could dynamically change this too. So we could have a different `terraform.tfvars` file per department for example. Allowing us to vary the instance sizes used for different environment types by department.

Type constraints - Simple Types

So far we have been setting variables and not specified the type. This means that whatever type you pass to a variable will be the type that the variable assumes. A type constraint allows you to specific the type of a variable. There are 3 simple types `string`, `number` and `bool`.

To specify a type constraint use the `type` property when defining a variable. Lets see an example of using the 3 simple types. Create a new folder and a file called `main.tf` and paste in the following code (or grab the code from `variables_example_05` if you are using the examples repository):

```
variable "a" {
  type    = string
  default = "foo"
}

variable "b" {
  type = bool
  default = true
}

variable "c" {
  type = number
  default = 123
}

output "a" {
  value = var.a
}

output "b" {
  value = var.b
}

output "c" {
  value = var.c
}
```

In this project we are defining 3 variables `a`, `b` and `c`. We are using the type parameter to define `a` as a `string`, `b` as a `bool` and `c` as a number. We are using defaults to set initial values for these variables so we do not have to worry about setting them using one of the techniques described earlier is this chapter and then we are outputting them by using outputs so we can see the values of them.

By using a type constraint you make it illegal to set the variable to anything other than the type defined. So for example if you try and set `b` to `"hello"` or `123` and run Terraform then Terraform will print an error saying that the value you have provided is not compatible with the type constraint.

There are a few interesting quirks with how the value you give is interpreted that are worth knowing. When you define the type to be `bool` then the following values will be valid `true`, `false`, `"true"`, `"false"`, `"1"` (evaluated to true), `"0"` (evaluated to false). The interesting part is that `"1"` is valid but `1` (without the quotes) is not valid. Which may be puzzling at first. The reason that this is the case is that when you define a string (by using quotes) then Terraform will attempt to evaluate what is inside that string whereas if you do not use quotes then Terraform will just see `1` as a number and not attempt to convert it to a bool. With a `number` any valid number will be allowed with or without quotes. If you specify quotes then you are essentially defining a string but as long as the string only contains digits, Terraform will realise this and implicitly convert the value to a number. With a string any value in quotes will be allowed.

The fact that Terraform coalesces variables (attempts to see what a variable is) is one of the reasons that its a good idea to use a type constraint. Another reason is that it guides the person using the Terraform code as to what value to use for the variable and eases code readability.

As well as the 3 simple types above these types can be combined into the following more complex types:

- list(<TYPE>)
- set(<TYPE>)
- map(<TYPE>)
- object()
- tuple([<TYPE>, ...])

For each of the complex types you can use another complex type type or simple type where you see the word `<TYPE>`. So you can have a list of number or a list of string or a list of map of string.

Type constraints - List

A list is a list of a type. So you can have a list of strings like `["foo", "bar"]` or a list of number `[2, 4, 7]`. The type means that every element in the list will be that type.

Create a new folder and file called `main.tf` and paste in the following code (or copy the code from `variables_example_06` in the examples repo):

```
variable "a" {
  type    = list(string)
  default = ["foo", "bar", "baz"]
}

output "a" {
  value = var.a
}

output "b" {
  value = element(var.a, 1)
}

output "c" {
  value = length(var.a)
}
```

In the above code we are defining a list in variable `"a"`. We are defining the list to be a list of string. To do this we use the word `list` and then put the type we want in brackets. Then to set the value for a list, you put the elements in between square brackets.

In the output variables the output variable `"a"` is simply printing the list. Variable `"b"` is using the inbuilt function `element` that can operate on lists. The HCL language provides various inbuilt functions to allow you to manipulate data. The `element` function takes a list as its first argument and a number for its second argument. The function will return the item at the element given. Because lists in HCL start from element `0` the value of output `b` will be `"bar"`. In output `"c"` we are using another inbuilt function `length`. The `length` function takes a list and returns the length of the list, so the `"c"` will output `3`. You can verify the output values by running the project by doing `terraform apply`. Play with the values so you get a feel for how lists work and how the functions work.

Type constraints - Set

A set is almost exactly the same as a list, the key difference is that a set only contains unique values.

Create a new folder and file called `main.tf` and paste in the following code (or copy the code from `variables_example_07` in the examples repo):

```
variable "my_set" {
  type    = set(number)
  default = [7, 2, 2]
}

variable "my_list" {
  type    = list(string)
  default = ["foo", "bar", "foo"]
}

output "set" {
  value = var.my_set
}

output "list" {
  value = var.my_list
}

output "list_as_set" {
  value = toset(var.my_list)
}
```

In the above example we are defining a variable called `my_set` and initialising it to the set `[7, 2, 2]` as I said above a set only contains unique values. So when you run this project by doing `terraform` apply you will set that the output value set will print [7, 2]. Terraform removes one of the 2 values as it was a duplicate. To show how sets can be useful we are defining a list called my_list where we are repeating the value foo twice. The value of the list output will be ["foo", "bar", "foo"] this is because we are just outputting the value of my_list which is of type list and lists can contain duplicate values. For the output list_as_set we are using the toset function to convert the my_list variable to a set. The value of this output will be ["foo", "bar"]. Because we are converting the list to a set Terraform removes the duplicate value "foo".

Type constraints - Tuple

A tuple are a strongly typed collection of one or more values. So for example you could define a tuple of three values string, number, number or two values string, string. Once a tuple is defined it always has to contain the number of values as defined in that tuple. The values also have to be the correct type and in the correct order based upon type.

Create a new folder and file called main.tf and paste in the following code (or copy the code from variables_example_08 in the examples repo):

```
variable "my_tup" {
  type    = tuple([number, string, bool])
  default = [4, "hello", false]
}

output "tup" {
  value = var.my_tup
}
```

In the above Terraform code we are defining a tuple variable my_tup with 3 values number, string, bool. The syntax for defining a tuple uses the form tuple([TYPE, TYPE...]). As stated above because we are defining a tuple with 3 values we have to set it to a value with 3 values, hence why we are initialising it to 4, "hello", false. If you run this example by doing terraform apply you will see that the output looks exactly like a list. You can kind of think of a tuple as a defined list where each element will always have a certain type and it will always be of a set length.

Try adding another value to the default for my_tup e.g. set it to [4, "hello", false, "hey"]. If you try doing a terraform apply on this you will see that Terraform gives you an error. This is because the value you are giving the tuple no longer matches the definition.

Type constraints - Map

A map as we have already covered in this chapter is a set of values indexed by key name. You can give a map a type, the type will be the type of the values.

Create a new folder and file called `main.tf` and paste in the following code (or copy the code from `variables_example_09` in the examples repo):

```
variable "my_map" {
  type    = map(number)
  default = {
    "alpha" = 2
    "bravo" = 3
  }
}

output "map" {
  value = var.my_map
}

output "alpha_value" {
  value = var.my_map["alpha"]
}
```

In this project we are creating a map of type `number`. We are initialising the map to have two keys `alpha` and `bravo`, the values for the keys are `2` and `3` respectively. The fact that we have specified that the map is of type `(number)` means that all of the values have to match the number constraint.

The `map` output value is going to print the whole map. We are also selecting a value out of a map by key using the `[]` syntax (as we have done previously in this chapter). The `alpha_value` output will print the value for the `alpha` key in the map which will be `2`. Feel free to run the project by using `terraform apply` and get a feel for how the code is working.

Type constraints - Object

An object is a structure that you can define from the other types listed above. It allows you to define quite complex objects and constrain them to types. I think the easiest way to explain it is to dive straight into an example.

Create a new folder and file called `main.tf` and paste in the following code (or copy the code from `variables_example_10` in the examples repo):

```
variable "person" {
  type = object({ name = string, age = number })
  default = {
    name = "Bob"
    age  = 10
  }
}

output "person" {
  value = var.person
}

variable "person_with_address" {
  type = object({ name = string, age = number, address = object({ line1 = string, li\
ne2 = string, county = string, postcode = string }) })
  default = {
    name = "Jim"
    age = 21
    address = {
      line1 = "1 the road"
      line2 = "St Ives"
      county = "Cambridgeshire"
      postcode = "CB1 2GB"
    }
  }
}

output "person_with_address" {
  value = var.person_with_address
}
```

In the project above we are are first defining a variable called `person`. This variable has two fields a `name` which is of type `string` and an `age` which is of type `number`. To define the object we specify each field inside `{}` brackets. Each field has the form `fieldname = type`. We are giving person some default values. If you run this project by doing `terraform apply` you will see that the `person` output will contain the values we gave it `Bob, 10`. These values are constrained to the types give so if you tried to assign a `string` to the `age` field you would get a type constraint error. One interesting thing to point out here is that you are allowed to have different items with the same name in Terraform. In this project we have a `variable` with the identifier person and we have an output with the same identifier. This is allowed because one is a variable and the other is an output. You are not allowed to have two variables with the same identifier or two outputs.

Next we are defining a variable called `person_with_address` to show how you can nest objects to build up even more complex structures. The person structure is the same as before except we have added the field `address`. The field `address` is in itself an object which contains four strings `line1, line2, county, postcode`. You can see when we initialise the variable we set the address by wrapping the values in a set of `{}` brackets. When you run the project you will see the `person_with_address` structure printed out.

By using the same technique as above you can build up structures which are arbitrarily complex.

Type constraints - Any type

The `any` type is a special construct that serves as a placeholder for a type yet to be decided. `any` itself is not a type, Terraform will attempt to calculate the type at runtime when you use `any`.

Lets go into an example of using the `any` type. Create a new folder and file called `main.tf` and paste in the following code (or copy the code from `variables_example_11` in the examples repo):

```
variable "any_example" {
  type = any
  default = {
    field1 = "foo"
    field2  = "bar"
  }
}

output "any_example" {
  value = var.any_example
}
```

In the above example we are defining an object with the `any` type. Because we are initialising the object to a default value, Terraform will use this default value to figure out the type of our variable `any_example`. Terraform will give our variable the type `object({ field1 = string, field2 = string})`. We are then printing out the `any_example` variable using an output.

Chapter 11 - Project Layout

Layout

Up until now we have been creating several Terraform projects to get used to different concepts we have been learning. But in every project we have been creating a single file called `main.tf` where we have placed all of our Terraform code (HCL). You may have wondered why all of the code was in a single file and what was the significance of that? Terraform actually does not care what the name of the file is, as long as it ends in `.tf`. So in all of the projects so far we could have called the file `project.tf`, `code.tf` or `foo.tf`. It really does not matter.

We can also split the code over as many files as we wish. The only rule is that all of the files have to be in the same folder because folders have a significance in Terraform (as we will learn in the next chapter on modules). The top level folder we have been working in is considered the main Terraform project. All files at the level (directly in that folder) are considered to be part of the project.

So if we have the following folder structure on disk:

```
/project
    main.tf
```

Then inside `main.tf` we had the code as we had in our first Terraform project:

```
provider "aws" {
  region = "eu-west-1"
}

resource "aws_s3_bucket" "first_bucket" {
  bucket = "<yourname>-first-bucket"
}
```

Then Terraform will consider that we are setting up the AWS provider and creating a single S3 bucket. Now if we change our project folder structure to the following:

```
/project
    provider.tf
    s3.tf
```

Then in the `provider.tf` file we place the following code:

```
provider "aws" {
  region = "eu-west-1"
}
```

In `s3.tf` we place the S3 bucket creation:

```
resource "aws_s3_bucket" "first_bucket" {
  bucket = "<yourname>-first-bucket"
}
```

Then if we ran Terraform again with the files in this structure Terraform would not show any difference. Terraform would still create a single S3 bucket. A good way to think about this is that when you run Terraform, it combines all of the code from all of the files in the top level folder together (in this case `/project`) and makes them into a single big text block. It then works out what you have defined in that block and the original filenames are thrown away.

Even though this means we are free to name the files and organise our code as we wish there are some conventions the community follow that make it easier to move from one Terraform project to another. Generally providers are setup in a file called `main.tf`. This gives you a place to look as to where all providers are configured. Files are then normally broken up around different areas of the system. For example if we are creating an AWS ECS cluster, we could put all of the setup for that cluster in a file called `ecs.tf`. If we are configuring our DNS entries in route53 we could put that in a file called `dns.tf`. The important point is to layout the code as to how it makes the most sense for you and the people you are working on the project with.

One last thing to cover on project structure is what happens when you create a child folder and put some code in there? Lets update the project above and add a child folder so our project structure now looks like:

```
/project
    provider.tf
    s3.tf
    other/
        bucket.tf
```

Then inside the file `bucket.tf` place the following code:

```
resource "aws_s3_bucket" "other" {
  bucket = "<yourname>-other-bucket"
}
```

If you run Terraform by running `terraform init` and then `terraform apply` from inside the `project/` folder you will see the Terraform runs and creates one S3 bucket. It completely ignores the code inside `bucket.tf`. That is because it is in a sub folder and only the code in the top level folder is considered by Terraform. *Note if you want to get the code for the last example then it is in the folder `project_layout_example_01` in the examples repository.*

Chapter 12 - Modules

Modules

Last chapter we learned about how Terraform handles files and folders in a project. You may have wondered what is the purpose of creating child folders if Terraform ignores them? It is because Terraform uses child folders to allow you to create modules.

A module in Terraform is a mini Terraform project that can contain all of the same constructs as our main Terraform project (resources, data blocks, locals, etc). Modules are useful as they allow us to define a reusable block of Terraform code and have many instances of it in our main Terraform project.

Modules in action

Lets go into an example of how a module works. For this example the code and directory structure will be listed as with a module you need to have a nested folder. If you wish to copy the code already written then you can get it from the `modules_example_01` folder in the examples repository.

Folder structure:

```
modules_example_01/
    sqs-with-backoff/
        main.tf
        variables.tf
        output.tf
    main.tf
```

Code:

top level `main.tf`:

```
provider "aws" {
  region  = "eu-west-1"
  version = "~> 2.27"
}

module "work_queue" {
  source      = "./sqs-with-backoff"
  queue_name  = "work-queue"
}

output "work_queue_name" {
  value = module.work_queue.queue_name
}

output "work_queue_dead_letter_name" {
  value = module.work_queue.dead_letter_queue_name
}
```

sqs-with-backoff/main.tf:

```
resource "aws_sqs_queue" "sqs" {
  name                       = "awesome_co-${var.queue_name}"
  visibility_timeout_seconds = var.visibility_timeout
  delay_seconds              = 0
  max_message_size           = 262144
  message_retention_seconds  = 345600 # 4 days.
  receive_wait_time_seconds  = 20 # Enable long polling
  redrive_policy             = "{\"deadLetterTargetArn\":\"${aws_sqs_queue.sqs_dead_\
letter.arn}\",\"maxReceiveCount\":${var.max_receive_count}}"
}

resource "aws_sqs_queue" "sqs_dead_letter" {
  name                       = "awsome_co-${var.queue_name}-dead-letter"
  delay_seconds              = 0
  max_message_size           = 262144
  message_retention_seconds  = 1209600 # 14 days.
  receive_wait_time_seconds  = 20
}
```

sqs-with-backoff/output.tf:

```
output "queue_arn" {
  value = aws_sqs_queue.sqs.arn
}

output "queue_name" {
  value = aws_sqs_queue.sqs.name
}

output "dead_letter_queue_arn" {
  value = aws_sqs_queue.sqs_dead_letter.arn
}

output "dead_letter_queue_name" {
  value = aws_sqs_queue.sqs_dead_letter.name
}
```

sqs-with-backoff/variables.tf:

```
variable "queue_name" {
  description = "Name of queue"
}

variable "max_receive_count"{
  description = "The maximum number of times that a message can be received by consu\
mers"
  default = 5
}

variable "visibility_timeout" {
  default = 30
}
```

We have written quite a lot of code. Lets dive into it piece by piece and explain what it is all doing. In a module you can take arguments. This allows you to give the user a chance to specify things about this instance of a module. The module that we have written is a module that creates two AWS SQS queues. One of the queues is a dead letter queue of the other. For our module we are allowing the user to specify the name of the queue. We are doing this by defining the variable `queue_name`. Variables have a special meaning when used with a module, they are the input values for your module. Note that inside a module Terraform does not care what the filenames are as long as they end in `.tf`. However, there is a convention where variables go in a file called `variables.tf` so we are going to stick with that. As the `queue_name` variable does not have a default a value must be given when the module is used. Variables in modules at in the same way as they do at the top level as we learnt in chapter 10, if you do not provide a default value then you have to provide a value for the variable when using the module. As queue_name does not have a default, it is a required parameter that must be set when using this module. The other two variables we have defined max_receive_count and visibility_timeout have defaults, so they do not have to be given values when using the module.

Next lets look at the main.tf file inside the module. You can see that we are using the passed in variables by using the var.<variable_name> syntax. The other interesting thing about the code is that we are setting the redrive_policy of the first queue to have a dead letter queue arn of the second queue. We are doing this using the interpolation syntax we learnt earlier. This allows Terraform to dynamically put the arn from the second queue in the redrive policy when it creates the first queue.

The last file that makes up our module is output.tf. This is where our outputs (or return values) from the module are specified. It is optional to return values out of a module but most modules do return values as it makes them easier to use. Outputs are defined using the output blocks that we learnt back in Chapter 7. Return values in a module can be used in the main Terraform project.

Look at the top level main.tf file and we can see how we create an instance of a module:

```
module "work_queue" {
  source      = "./sqs-with-backoff"
  queue_name  = "work-queue"
}
```

To create an instance of a module we start with the keyword module. You then follow that with the identifier you want to use to refer to that instance of the module. You then surround the module block in { and }. Inside the module every module has a property called source. The source property is where the code is for the module. You can see that we are using the local path ./sqs-with-backoff. This is telling Terraform that it can find the code for this module in a local folder with that name. We are then giving a value of work-queue to the queue_name property. Note that we are not specifying the max_receive_count or visibility_timeout. We do not need to as they both have default values. If we wanted to we could give values for them and then they would be used instead of the defaults.

At the bottom of the main.tf we are outputting the values of the names of the two queues that are created in the module. This is working because we are referencing the values returned by the module. To reference a value returned by a module, you use the following syntax module.<module_identifier>.<output_name>. So to reference the value of the main queue you would use module.work_queue.queue_name. The keyword module is constant.

Open a terminal inside the project and run terraform init and terraform apply then confirm by typing yes. You will see that two AWS SQS queues get created. One is a dead letter queue of the other and the queue names are printed to the console when Terraform finishes running.

The real power of modules is that it allows you to put logic such as this in a single place and then reuse it across your Terraform project. You can have multiple instances of a module in a single Terraform project. If you add the following code to the top level main.tf in your project (or code can be found in modules_example_02):

```
module "thread_queue" {
  source     = "./sqs-with-backoff"
  queue_name  = "thread-queue"
}
```

Above we are defining a second instance of a the same module. If you run the project now (`terraform apply`) you will see that Terraform creates two more queues one being a dead letter of the second. This is a good way to write code because if we want to change something about the way our queues are constructed we can change it in a single place and then run `terraform apply` and all of instances of the module will get this change. For example try changing the `max_message_size` property on one of the queues inside the module and run Terraform again. You will see that both queues get updated. If you would have written this code by having all of the SQS resources in the top level Terraform project then you would have had 4 Terraform resources defined (one for each queue) and if you wanted to make a change you would have had to update each resource with that change. That would quickly become unmanageable if you had tens or even hundreds of instances of your module.

Using modules also allows you to define a structure of how items should look. In our example we prefixed our queue names with `awesome_co-`. This is a great technique if you want to name all of your queues in a certain way. You could optionally take a variable which is the name of the environment and then the module will prefix the name of the environment onto the queue name so the queue names will not clash.

Returning a complex type from a module

It is possible to return a whole resource from a module. This means we can give all of the fields back from the resource that we created which then lets the user use whichever values that they want. To see this in action lets update the sqs backoff example that we just worked on (or copy the folder `modules_example_02` from the examples repository).

Start by updating the `output.tf` file inside the `sqs-with-backoff` directory to the following:

```
output "queue" {
  value = aws_sqs_queue.sqs
}

output "dead_letter_queue" {
  value = aws_sqs_queue.sqs_dead_letter
}
```

What we are doing is setting the `value` of the `queue` output to the `aws_sqs_queue.sqs` SQS queue resource itself. This is saying return every attribute of that resource from the `queue` output. We are then doing the same with the `dead_letter` output for the dead letter queue resource. Next update the `main.tf` by removing all of the outputs and replacing them with:

```
output "work_queue" {
  value = module.work_queue.queue
}

output "work_queue_dead_letter_queue" {
  value = module.work_queue.dead_letter_queue
}
```

Run the terraform project again you will see that all of the attributes of both queues are now printed to the console. As all of the attributes are being returned as outputs from both of the queues the consumer of the `sqs-with-backoff` module is free to use any of those attributes in the rest of their Terraform project.

Modules using a sub module

Modules can themselves use modules inside them. We are going to go through an example of a 3 way cross talk module. The purpose of this module is to setup ingress and egress on a protocol and port of your choosing between 3 AWS security groups in both directions. To do this by hand would require 12 security group rules but this can be shortened by using modules.

Lets dive straight into the example. If you are following along with the book then create a folder in your work space and add the following file structure (or if you are using the examples repository then the code can be found in the folder `modules_example_03`):

Folder structure:

cross-talk/
main.tf
variables.tf
cross-talk-3-way/
main.tf
variables.tf
main.tf

cross-talk/main.tf:

```
resource "aws_security_group_rule" "first_egress" {
  from_port                 = var.port
  to_port                   = var.port
  protocol                  = var.protocol
  security_group_id         = var.security_group_1.id
  type                      = "egress"
  source_security_group_id  = var.security_group_2.id
}

resource "aws_security_group_rule" "first_ingress" {
  from_port                 = var.port
  to_port                   = var.port
  protocol                  = var.protocol
  security_group_id         = var.security_group_1.id
  type                      = "ingress"
  source_security_group_id  = var.security_group_2.id
}


resource "aws_security_group_rule" "second_egress" {
  from_port                 = var.port
  to_port                   = var.port
  protocol                  = var.protocol
  security_group_id         = var.security_group_2.id
  type                      = "egress"
  source_security_group_id  = var.security_group_1.id
}

resource "aws_security_group_rule" "second_ingress" {
  from_port                 = var.port
  to_port                   = var.port
  protocol                  = var.protocol
  security_group_id         = var.security_group_2.id
  type                      = "ingress"
  source_security_group_id  = var.security_group_1.id
}
```

cross-talk/variables.tf:

```
variable security_group_1 {}
variable security_group_2 {}
variable port {
  type = number
}
variable "protocol" {}
```

Lets pause and discuss the `cross-talk` module that we have just defined. Looking at the `variables.tf` we have introduced a couple of new concepts. The first is that we are taking the whole security group resource as an input. This was introduced in Terraform `0.12` and makes writing a module much easier. As it means you can take the whole resource and then pick off the properties you need from it. Rather than having to take all of the fields you need as separate properties, as was the case in Terraform pre `0.12`.

The next new concept is constraining a variable to a type. As the `port` has to be a number we can constrain it by using `type = number`. This gives the user of our module some guidance so they know from looking at the input that they must supply a number. It also means the IDE and tooling can prompt the user before they run Terraform that the variable must be a number.

Inside the `cross-talk/main.tf` we are defining a security group ingress and egress pair of rules for each security group passed in. This allows the traffic between them on the protocol provided in both directions. This module is useful in itself in that it allows 2 way talking on a protocol and port for 2 security groups in AWS.

cross-talk-3-way/main.tf:

```
module "first_to_second" {
  source             = "../cross-talk"
  security_group_1   = var.security_group_1
  security_group_2   = var.security_group_2
  protocol           = var.protocol
  port               = var.port
}

module "second_to_third" {
  source             = "../cross-talk"
  security_group_1   = var.security_group_2
  security_group_2   = var.security_group_3
  protocol           = var.protocol
  port               = var.port
}

module "first_to_third" {
  source             = "../cross-talk"
  security_group_1  = var.security_group_1
  security_group_2  = var.security_group_3
  protocol          = var.protocol
  port              = var.port
}
```

cross-talk-3-way/variables.tf:

```
variable security_group_1 {}
variable security_group_2 {}
variable security_group_3 {}
variable port {
  type = number
}
variable "protocol" {}
```

The next module `cross-talk-3-way` has one extra variable (`security_group_3`) to take the third security group resource. It then simply has `3` instances of the `cross-talk` module. As we have `3` modules then there are `3` pairs that we need to open up rules between. By using a module to define a cross talk between two security groups and then another module to define cross talking between three, we can keep our Terraform code neat and compact. This is a good example where having sub modules makes the code quite readable.

However, often using too many sub modules can make your code confusing and overly complex. So they are a matter of judgement. A good rule of thumb is not to go more than one level of nesting deep unless you have a really good reason like it makes the code much easier to read and understand. If that is not the case then it is probably not the place to use a module.

To finish our example define the following `main.tf` at the top level:

```
provider "aws" {
  region  = "eu-west-1"
  version = "~> 2.27"
}

resource "aws_security_group" "group_1" {
  name = "security group 1"
}

resource "aws_security_group" "group_2" {
  name = "security group 2"
}

resource "aws_security_group" "group_3" {
  name = "security group 3"
}

module "cross_talk_groups" {
  source           = "./cross-talk-3-way"
  security_group_1  = aws_security_group.group_1
  security_group_2  = aws_security_group.group_2
  security_group_3  = aws_security_group.group_3
  port             = 8500
  protocol         = "tcp"
}
```

In our `main.tf` we are configuring our AWS provider with `region` and pinned to a `version` (as per best practice as we have learnt). We are then defining 3 security groups. Lastly we are using the `cross-talk-3-way` module to setup cross talking between them on `tcp` on port `8500`. By using modules to implement the cross talk functionality our actual Terraform code (which is everything in the top level folder) is short and concise. If we wrote all of the code to implement cross talking between the 3 security groups at the top level we would end up with 12 security group rule resources. We would have to make sure that we got each of those exactly right. By using modules we can define the logic in a simpler and easier to understand way. As first we have a module that allows cross talking between 2 modules then that module is used to create a module that allows cross talking between 3 security groups. Then at the top level our code is much shorter and more readable.

Remote modules

Modules are great for enabling you to reuse blocks of configuration across a project but what if you want to build up a library of great modules and share them across your company or with your friends? Terraform have solved that with remote modules. A remote module is a module hosted externally to the local file system. Terraform supports many different remote module sources such as GitHub, BitBucket and S3

We are going to use GitHub to host the module `sqs-with-backoff` module that we declared and then reference it from our local project. Create a new folder and then paste the following into `main.tf` (or get the code from `modules_example_04` folder in the examples repository):

```
provider "aws" {
  region  = "eu-west-1"
  version = "~> 2.27"
}

module "work_queue" {
  source      = "github.com/kevholditch/sqs-with-backoff"
  queue_name  = "work-queue"
}

output "work_queue" {
  value = module.work_queue.queue
}

output "work_queue_dead_letter_queue" {
  value = module.work_queue.dead_letter_queue
}
```

You will notice that this code is almost the same as the code we wrote earlier when using the `sqs-with-backoff` module. The only difference is that we are setting the `source` of the module to `github.com/kevholditch/sqs-with-backoff` rather than the local path to the folder. By default Terraform will add on the `https://` on the front of the URL so we do not need to include that.

Initialise Terraform by opening a terminal and running `terraform init` from inside the folder where you have written the `main.tf` file. You will see at the top of the output something like the following:

```
Initializing modules...
Downloading github.com/kevholditch/sqs-with-backoff for work_queue...
- work_queue in .terraform/modules/work_queue
```

Terraform clones the code for the module when it is initialised. The code is downloaded to `.terraform/modules`. If you have a look in that folder you will see the module as defined in the repository `github.com/kevholditch/sqs-with-backoff`. As we have specified the URL to the repository the clone will be done from GitHub using https. If you want to instead clone using SSH then you can do that by changing the URL to `git@github.com:kevholditch/sqs-with-backoff.git` which is the SSH clone address. It is cool that Terraform allows you to clone the module either way. *Note you will need SSH setup with GitHub in order for the SSH clone to work.*

Run the project in by running `terraform apply` and you will see the same output as before where the two SQS queues are created, one being a dead letter queue of the other one.

In a team environment when you are using remote modules to share your modules using GitHub you want to ensure that every time Terraform is run that it always uses the same version of the module. You want to be able to consciously control when you move to a newer version of the module if one is checked in. By default with the code we have written above you will always get the latest version of the module every time you run Terraform, which is probably not what you want. As if someone makes a change to the module then that change will start rolling out the next time anyone runs a Terraform project that references the module.

To get around this problem you can pin the version of the remote module you reference by using a git tag. A tag in git is simply a marker to a commit. You can think of a git tag as a branch that never moves. In the repository `github.com/kevholditch/sqs-with-backoff` I have created a tag `0.0.1` which points to an older version of the module. If you update the source in our code example to `github.com/kevholditch/sqs-with-backoff?ref=0.0.1` (or get the code from `modules_example_05` in the examples repository) and run `terraform init` and then `terraform apply` again. You will notice that the SQS queue names are now prefixed with `bad_co` instead of `awesome_co`.

This happened because by adding the parameter `ref=0.0.1` we are telling Terraform to use the source code inside the GitHub repository from the commit that was tagged with `0.0.1`. The commit that the tag points to has the different queue prefix in, so you can see that something has changed. When you omit the `ref` parameter as we did at the start, you get the latest version from the `master` branch. This is almost never what you want, as otherwise your code will update any time someone makes a change. It is much better to be able to control when you move by using the `ref` parameter.

Now update the source url to `github.com/kevholditch/sqs-with-backoff?ref=0.0.2`. Run `terraform init` and `terraform apply` again. The SQS queue names now get put back to `awesome_co`. The tag `0.0.2` also points to the latest version of the `master` branch so the effect is the same as leaving the `ref` parameter off.

Chapter 13 - Plans

Plans

A plan is Terraform showing you how it needs to change the world to get it into the desired state specified in your code. Terraform plans, detail you what Terraform will create, what Terraform will destroy and what Terraform will update. This gives you a view on what Terraform is going to do before you ask Terraform to do it. Terraform summarises how many creates, updates and destroys it is going to do at the bottom of the plan.

So far in this book we have been running `terraform apply` which will automatically do a plan first and then pause and wait for you to confirm. When Terraform pauses it is giving you time to review the plan to make sure it is correct before executing it. In this chapter we are going to learn how to read plans and some key things to look out for.

Lets create a small project so we can explore how plans work. Create a new folder in your workspace and create a single file in the folder called `main.tf` and paste in the following code (or if you are using the examples from GitHub then the folder with the code is `plans_example_01`):

```
provider "aws" {
  region  = "eu-west-1"
  version = "~> 2.27"
}

resource "aws_sqs_queue" "test_queue" {
  name = "test_queue"
}
```

By now you should be able to understand what the HCL above is going to do, it is going to setup the AWS provider and create a queue with the name `test_queue`. Open your terminal and go into the folder where you have created the project and run `terraform init` to initialise Terraform and then `terraform apply` but do not confirm just yet.

You should see a plan that looks something like this:

```
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_sqs_queue.test_queue will be created
  + resource "aws_sqs_queue" "test_queue" {
      + arn                               = (known after apply)
      + content_based_deduplication       = false
      + delay_seconds                     = 0
      + fifo_queue                        = false
      + id                                = (known after apply)
      + kms_data_key_reuse_period_seconds = (known after apply)
      + max_message_size                  = 262144
      + message_retention_seconds         = 345600
      + name                              = "test_queue"
      + policy                            = (known after apply)
      + receive_wait_time_seconds         = 0
      + visibility_timeout_seconds        = 30
    }

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:
```

The plan above is going to create a single SQS queue (as we were expecting). Everything that Terraform is going to create is marked with a + symbol. The line `+ resource "aws_sqs_queue" "test_queue"` is showing that Terraform is going to create a new resource. Because of this every attribute is also marked with + which may be obvious when you are creating a new resource but the attribute markings will be more important later on.

At the bottom of the plan is the summary `Plan: 1 to add, 0 to change, 0 to destroy.`. In the summary Terraform adds up all of the resources it is going to create (add), update (change) and destroy. It is good practice to look at this line first and consider whether any of the numbers look completely wrong. For example if you are expecting to be creating some new resources and Terraform says `0 to add, 4 to change, 4 to destroy` then Terraform will not be creating anything and you will want to review the plan carefully.

The plan above is what we are expecting so lets confirm it by typing `yes` and pressing enter.

Once Terraform has applied those changes update the `aws_sqs_queue` resource to the following (code is in `plans_example_02` folder in the examples repository if you want to paste it in from there *note you will have to copy the code across you cannot use a new folder otherwise Terraform will consider it a different project*):

```
resource "aws_sqs_queue" "test_queue" {
  name                       = "test_queue"
  visibility_timeout_seconds = 45
}
```

Now run `terraform apply` again and you should see a plan like:

```
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:

  # aws_sqs_queue.test_queue will be updated in-place
  ~ resource "aws_sqs_queue" "test_queue" {
        arn                               = "arn:aws:sqs:eu-west-1:282637107404:test\
_queue"
        content_based_deduplication       = false
        delay_seconds                     = 0
        fifo_queue                        = false
        id                                = "https://sqs.eu-west-1.amazonaws.com/282\
637107404/test_queue"
        kms_data_key_reuse_period_seconds = 300
        max_message_size                  = 262144
        message_retention_seconds         = 345600
        name                              = "test_queue"
        receive_wait_time_seconds         = 0
        tags                              = {}
      ~ visibility_timeout_seconds        = 30 -> 45
    }

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

Notice this time the resource is marked with ~ which means the resource will be updated in place. An update in place means the resource can be updated without destroying it and recreating it. It is important to spot when an update can happen in place which is generally safe and when a resource has to be destroyed first and then recreated (we will cover this case soon). The ~ symbol is next to the `visibility_timeout_seconds` attribute showing that this attribute will be changed if you confirm this apply. You can see that the value is changing from `30` to `45`.

Apply this plan by typing `yes` and pressing enter. This will now modify the queue so that it has a visibility timeout of `45` seconds.

Update the `aws_sqs_queue` resource again with the following code (or you can get the code example from the folder `plans_example_03` in the examples repository *note you will have to copy the code across you cannot use a new folder otherwise Terraform will consider it a different project*):

```
resource "aws_sqs_queue" "test_queue" {
  name                       = "my_queue"
  visibility_timeout_seconds = 45
}
```

We are changing the name of the SQS queue from `test_queue` to `my_queue`. In AWS there is no way to change the name of a SQS queue once it is created. To do this you have to destroy your old queue and create a new one. Lets see how Terraform handles the update we have made to our code. Run `terraform apply` and view the plan which should look something like this:

```
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:

  # aws_sqs_queue.test_queue must be replaced
-/+ resource "aws_sqs_queue" "test_queue" {
      ~ arn                               = "arn:aws:sqs:eu-west-1:282637107404:test\
_queue" -> (known after apply)
        content_based_deduplication       = false
        delay_seconds                     = 0
        fifo_queue                        = false
      ~ id                                = "https://sqs.eu-west-1.amazonaws.com/282\
637107404/test_queue" -> (known after apply)
      ~ kms_data_key_reuse_period_seconds = 300 -> (known after apply)
        max_message_size                  = 262144
        message_retention_seconds         = 345600
      ~ name                              = "test_queue" -> "my_queue" # forces repl\
acement
      + policy                            = (known after apply)
        receive_wait_time_seconds         = 0
      - tags                              = {} -> null
        visibility_timeout_seconds        = 45
```

```
    }

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

The symbol next to the `aws_sqs_queue` is `-/+`. This is a special type of update where Terraform has to destroy your resource and then create a new one in order to make the change. It is important to note that destroy then creates are potentially dangerous as it means your resource may not be available for a period of time, so this type of change should applied with caution. At the bottom, the summary of the plan states `1 to add, 0 to change, 1 to destroy` which shows that we are destroying the resource and then recreating it, even though all we did was update the name of the queue.

When you apply the change above there will be a period of time where no SQS queue will exist. As what Terraform will do is first destroy the old queue and only once the queue is destroyed will it then create the new one. If you want Terraform to create the new queue before deleting the old one then this is possible using a resource lifecycle which will be covered in a later chapter. As a small aside if you want to change the name of an SQS queue then a better technique would be to create a new SQS queue in one release so both queues exist side by side, then switch your application over to using the new queue. Then do another Terraform release to delete the original queue.

Take the `main.tf` file and remove the `aws_sqs_queue` resource (or you can get the code example from the folder `plans_example_04` in the examples repository *note you will have to copy the code across you cannot use a new folder otherwise Terraform will consider it a different project*).

Run `terraform apply` and this time you will see a plan such as:

```
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_sqs_queue.test_queue will be destroyed
  - resource "aws_sqs_queue" "test_queue" {
      - arn                               = "arn:aws:sqs:eu-west-1:282637107404:my_q\
ueue" -> null
      - content_based_deduplication       = false -> null
      - delay_seconds                     = 0 -> null
      - fifo_queue                        = false -> null
      - id                                = "https://sqs.eu-west-1.amazonaws.com/282\
637107404/my_queue" -> null
      - kms_data_key_reuse_period_seconds = 300 -> null
      - max_message_size                  = 262144 -> null
      - message_retention_seconds         = 345600 -> null
      - name                              = "my_queue" -> null
      - receive_wait_time_seconds         = 0 -> null
      - tags                              = {} -> null
      - visibility_timeout_seconds        = 45 -> null
    }

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

As we have removed the SQS queue from our Terraform code you will see that Terraform wants to remove the SQS queue from AWS. The `-` symbol next to the `aws_sqs_queue` resource depicts the fact that Terraform wants to destroy the resource. The summary of the plan also shows that Terraform has 1 item to destroy.

Destroys and destroy then creates are the key thing to look out for when reviewing a Terraform plan. As both of them mean that you are going to destroy a resource. Once the resource is destroyed any state that was part of the resource could be lost and obviously if you delete a key part of your infrastructure you could cause an outage.

When reviewing a plan a good mindset to get into is to think about the changes you have made to your Terraform code and see if the plan matches what you are expecting. Start with the summary and see if you are seeing roughly the right amount of adds, updates and destroys and then work up from there. When changing a resource property carefully check whether that is going to result in Terraform needing to destroy and then recreate that resource in order to make that change. If you find the documentation for the resource it will say `force_create` next to the attributes that will cause a destroy then create if they are changed. This is what we saw in the SQS example we went through where we updated the name.

Plan command

Until now we have been taking advantage of the fact that when you run the command `terraform apply` by default it does a plan first and pauses for you to confirm before actually doing the apply. It is however possible just to get Terraform to do a plan without the option of applying it. To do this simply run the command `terraform plan`. A Terraform plan will produce a plan as we have seen before but it will not give you the option to apply it, instead it will just show you the plan. This command can be useful if you want to just test the water with something and do not want to run the risk of applying it by accident. It can also be useful when building Terraform deployment pipelines as we discuss below.

Lets go through an example of using the Terraform plan command. Go back to the example from the project `plans_example_01` in the code examples repository or use the following `main.tf` code:

```
provider "aws" {
  region = "eu-west-1"
}

resource "aws_sqs_queue" "test_queue" {
  name = "test_queue"
}
```

Open your terminal and go into the folder where the `main.tf` file is located and run `terraform plan`. Terraform will show you the same plan as we saw at the start of this chapter. You can save the plan that Terraform generates to a file by using the `-out` parameter. Lets try the plan again and this time save it to a file called `myplan` by using the command `terraform plan -out myplan`.

You may be wondering what you can do with the plan that you have saved to a file. Well you can use the plan you just saved with the apply command, to apply that plan to your infrastructure. Try running the following command `terraform apply myplan`. Terraform will just apply your plan saved in the file `myplan` without stopping to ask you if you wanted to do it or not. The reason being is that by running apply in this way you have broken the `terraform apply` command into two phases: the plan phase and the apply phase. Because you saved the plan to a file and you pass that plan to apply you are saying to Terraform that this is the plan I want you to execute in the apply.

Remember to finish up by destroying the resources by running `terraform destroy` and confirming by typing `yes` and pressing enter.

Being able to separate out the plan and apply phase enables you to write your own build deploy pipelines by having the plan as a step and then passing that plan file onto the apply step. This gives you the option to pause if you want and allow a human operator to check the plan before you pass it to be applied.

Auto apply

It is possible to skip the confirmation where Terraform pauses during the apply phase a different way, you can use the `-auto-approve` flag. Lets test this out by ysing the same Terraform code again (from the repository folder `plans_example_01`) run the command `terraform apply -auto-approve`. Terraform will simply create the AWS SQS queue without stopping to ask you to confirm. *Remember to destroy these resources once you have run the example.*

I wanted to show the `-auto-approve` flag in order to highlight a key difference between using that and passing a plan to the `apply` command which also causes apply to run without asking for you to confirm. The difference is that when you pass a plan to the apply command, the apply command will execute that exact plan. So if something has changed in your infrastructure and that plan is no longer possible then Terraform will error and tell you. It is a safe guard where you are explicitly saying to Terraform, go and execute exactly this plan. Whereas with the auto apply flag you are saying to Terraform go and work out what to do, generate a plan and then just go and do it without asking you to confirm. For this reason using the `-auto-approve` flag is much more dangerous and not really recommended in most scenarios, as if Terraform is going to destroy your database it is just going to go ahead and do it without asking you to confirm.

This is the reason that if you are building Terraform into a build deploy pipeline you want to use a plan file and pass it to apply. As it allows you to review the plan and then know for sure that Terraform will only be executing exactly the plan that you reviewed.

Chapter 14 - State

State

State is Terraform's store of all of the resources it has created. State stores all of the information about the resources, including meta information that cannot be retrieved from the underlying infrastructure APIs. It also stores the dependency order of the resources that it created. Terraform uses its state to work out how it needs to make changes. By default Terraform stores state in a local file called `terraform.tfstate`. You may have noticed this file in the examples we have been doing up until now.

A common thing that people wonder is that why does Terraform need to record the resources it has created in a state file, why can't it just look at the HCL code and compare that to the real world and apply the changes. There are quite a few reasons why Terraform would not work without its state file.

Some resources that you create have write once values, once they are written there is no way to retrieve them again. One such example is an RDS instance password. Imagine you created an RDS instance with a password `password123`, Terraform would create the RDS instance when you run it and store the password in its state file. Then if you change the password to `myPassword` in your HCL code, Terraform would know to update the RDS instance because it can compare the password in the state file to the one in the HCL code. If Terraform did not have its state file it would have no way of knowing that you had changed the password as you cannot retrieve it back from the AWS API.

The state file is also needed by Terraform for deleting resources. A simple example of this is that if you had a AWS subnet and an EC2 instance in that subnet and you deleted both of them from your project. Firstly Terraform needs to know that it created those resources in the first place. Terraform cannot assume that anything in AWS that is not in your HCL code was something that it created that can now be deleted. As you want the option to have infrastructure that Terraform is not managing. Terraform also needs to know to delete the EC2 instance before the subnet because the instance depends on the subnet. When you create the resources Terraform finds out that the EC2 instance depends on the subnet due to you referencing a subnet's output property as in input value into the EC2 instance subnet field. So it knows to create the subnet first. It then stores this dependency information in its state file. Terraform can then use the dependency order in its state file to delete the resources in the correct order. The only other way to solve this problem without a state file would be for Terraform to know the dependency order of all resource types. This quickly explodes to a point where it becomes almost impossible to maintain, plus it doesn't scale. Terraform uses state to elegantly sovle the dependency problem. That's a big part of the reason that it is so important to make sure you reference resources in their dependencies in your HCL code. For some resources it is almost impossible not to do this as you cannot guess certain values before AWS creates them such as an ID for a VPC. Some resource values can be guessed before they are created such as the ARN of a resource. So even when you could compute the ARN and just set it using a computed value, do not do this. Instead always use the output property for the resource you want to reference.

By default Terraform uses a combination of its state and the underlying infrastructure APIs to build a plan of how to make changes to get the world to how you have specified it in your HCL code. When projects reach a certain size and due to rate limits imposed on some APIs it can become impractical to interrogate them on every Terraform run. In this situation you can pass in a flag to Terraform to tell it just to use its state as the source of truth. This greatly speeds up the Terraform run time and gets round this problem. The downside being that if someone modifies the underlying infrastructure outside of Terraform then the Terraform run could fail as it will be unaware of these changes.

Manipulating State

It is important to understand how to manipulate Terraform state. Terraform state can be changed to enable you to do a number of operations such as import existing infrastructure that was created by hand into Terraform, move resources from one Terraform project to another and correct Terraform if someone goes and manually changes infrastructure behind Terraform's back where there is no way for Terraform to automatically know how to resolve that situation.

Lets learn how to import infrastructure into Terraform that was created outside of Terraform. First log into AWS and go to VPC. Click the `Create VPC` button, enter `example` in the Name tag field and `10.0.0.0/16` as the CIDR block then click the `Create` button. This will create the new VPC.

To get Terraform to manage this VPC we need to import it into Terraform. To do this create a new folder and create a file called `main.tf` and paste in the following code (or you can use the code from the folder `state_example_01` if you are using the GitHub examples repository).

```
provider "aws" {
  region  = "eu-west-1"
  version = "~> 2.27"
}

resource "aws_vpc" "example" {
  cidr_block = "10.0.0.0/16"

  tags = {
    Name = "example"
  }

}
```

The code above simply sets up the AWS Terraform provider and will create a new VPC with the CIDR block `10.0.0.0/16` and with the name tag `example`. The same as how we set it up in the UI. If you run Terraform apply now from this folder then Terraform will create another VPC. Give it a try. The reason for this is that the VPC you created manually is nothing to do with Terraform. Terraform only keeps track and manages resources that it has created itself. It ignores everything else. To get Terraform to manage your existing infrastructure you need to import it. If you have run `terraform apply` and created a second VPC then destroy it by running `terraform destroy` and confirm the destroy by typing `yes` when prompted.

To import the VPC you created by hand into Terraform go to the AWS UI and copy the VPC ID. Go to the command line and type `terraform import aws_vpc.example <VPC_ID>` replacing `<VPC_ID>` with the ID of the VPC that you got from AWS. You should see Terraform say `Import Successful`. Now if you run `terraform apply` Terraform will report `No changes. Infrastructure is up-to-date.`.

The Terraform import command told Terraform to take ownership (or import) the VPC that we created in AWS. This causes Terraform to go up to AWS read the resource and put it into its state. Remember the state is Terraform's store of what it created or which resources it manages. So by doing an import you are instructing Terraform to start managing that resource. The Terraform import command itself always takes the form `terraform import <resource_type>.<resource_identifier> <value>`. Where `<resource_type>` is the type of the resource you are importing, `<resource_identifier>` is the identifier you gave the resource. `<value>` can be an id or an identifier that Terraform can use to go and get the resource. The `<value>` field is different for every resource. To find out what to use in the import command consult the Terraform documentation for the resource that you are using. It is also worth noting that some resources cannot be imported. The provider author has to implement the import operation for the resource. However pretty much all most resources in the common providers (such as the AWs provider) do have import implemented.

Moving a resource from one project to another

When Terraform projects become large or you want to refactor them, a scenario that often comes up is that you want to move a resource from one Terraform project to another. To do this you cannot simply delete the resource from Terraform project and add it to the other as if you do that then Terraform will physically delete the resource and then recreate it again in the other project. This is often not desirable as it will cause loss of availability and/or data.

To solve this problem we are going to learn how you can manipulate the Terraform state to move resources between projects. Start by creating one folder, create a file called `main.tf` and paste in the following code (or if you are using the code from the examples repository then the code can be found in the folder `state_example_02a`):

```
provider "aws" {
  region  = "eu-west-1"
  version = "~> 2.27"
}

resource "aws_vpc" "my_vpc" {
  cidr_block = "10.1.0.0/16"

  tags = {
    Name = "vpc"
  }
}
```

Next create another folder, this will be our second Terraform project where we are going to move the resource to. Create a file called `main.tf` and paste in the following code (or if you are using the code from the examples repository then the code can be found in the folder `state_example_02b`):

```
provider "aws" {
  region  = "eu-west-1"
  version = "~> 2.27"
}

resource "aws_vpc" "main" {
  cidr_block = "10.1.0.0/16"

  tags = {
    Name = "vpc"
  }
}
```

Both of the projects are pretty similar. They both setup the AWS provider in the same way. They both define an AWS VPC with the same CIDR range (10.1.0.0/16) and both have a single name tag `vpc`. The VPCs do have a different identifier though, in the first project we are using the identifier `my_vpc` and in the second project the identifier is `main`.

Run the first project by opening your terminal and changing directory into the folder. Run `terraform init` and then run `terraform apply`. Confirm the apply by typing `yes` and pressing enter. We now have created our VPC. In this example we want to move this VPC to the second Terraform project but we do not want to destroy it. If you simply remove the block `my_vpc` from the project and run `terraform apply` then Terraform will want to destroy the VPC. This is not what we want. To make Terraform stop managing the resource we have to remove it from Terraform's state.

To remove the VPC from the Terraform's state we need to use the `terraform state rm` command. This command needs to be handled with care. To remove the VPC from the state run `terraform` `state rm aws_vpc.my_vpc`, you will see Terraform say `Removed aws_vpc.my_vpc`. This now means that the VPC will not be in Terraform's state any more, which means as far as Terraform is concerned it did not create the VPC and is not managing it. Note the VPC still exists in AWS (which is what we wanted). The format of the Terraform `state rm` command is `terraform state rm <resource_name>.<resource_idenitifier>`.

Now that the VPC is removed from our first project go ahead and delete the `aws_vpc.my_vpc` block from the project and run `terraform apply`. Terraform will report that there is nothing to do. The reason for this is that Terraform does not know about the VPC anymore. This is the affect the state removal had. We have now successfully removed the VPC from this project but crucially it still exists in AWS.

Next we want to make project 2 own the VPC, go into the directory for project 2. Run `terraform init` to initialise Terraform and then open the AWS console in a web browser and go to the VPC section. Find the ID of the VPC that we created using the first Terraform project and copy it to your clipboard. Now open a terminal in the directory for project 2 and run `terraform import aws_vpc.main <VPC_ID>`. This will import the VPC into the state of Terraform. Now run `terraform apply` and Terraform should report that there is nothing to do. This is because Terraform in project 2 is now managing the VPC and it matches our HCL code. Another interesting aside is that we changed the identifier of the VPC in our HCL code from `my_vpc` to `main`.

We have now successfully moved the VPC from one Terraform project to another. Although this was a bit of a trivial example, this technique can be used to break up larger Terraform projects when they become more cumbersome to manage.

There is another way to move a resource from one project to another and that is to use the `terraform state mv` command.

Lets use the `terraform state mv` command to move the AWS VPC from project 2 back to project 1. Before starting this exercise paste back the VPC block back into project 1 with the identifier `my_vpc`. Open your terminal inside the second Terraform project and run the command (changing the folder name for the first project where I am using `state_example02a` for the folder you are using):

```
terraform state mv -state-out=../state_example_02a/terraform.tfstate aws_vpc.main aw\
s_vpc.my_vpc
```

This command says move an item from Terraform state into the file specified by the flag `-state-out`. We are pointing that flag to the Terraform state of Terraform project 1. The next two arguments are the source resource and the destination resources. Remember we are using the identifier `main` in the second project and `my_vpc` in the first project. Run this command then go into the folder of the first project. If you run `terraform apply` you will see Terraform report that there is nothing to do. This command is has slightly different semantics from the remove and then import flow that we covered. This command removes the resource from the first project's Terraform state which is the same but rather than importing the resource it simply puts it directly into the state file. An import under the covers works slightly differently in that Terraform queries the API of the infrastructure to read the resource and uses that to build the state, with a move command the state is copied across directly.

The advantage of the `state mv` command is that it works on any resource even if it does not support an import.

One last thing I want to cover in this section is listing Terraform state. In the first project where we now moved the VPC back to, run the command `terraform state list`. This command lists all of the resources that exist in Terraform's state file. You should see the following output:

```
aws_vpc.my_vpc
```

The `terraform state list` command is handy if you want to interrogate Terraform to find out which resources are in its state file.

Remote state

As stated at the start of this chapter by default state is stored in a local file called `terraform.tfstate`. This is fine when you are working on a local project or for small proof of concepts but it does not really scale beyond that. If you want to work on a Terraform project on more than one machine or with more than one person so hence more than one machine, then local state won't cut it anymore. The reason being (as we have learnt in this chapter) is that Terraform uses its state file to store a record of what it has created so if you run Terraform from one machine and create a bunch of resources then run the same project from another machine, then Terraform will try and create all of those resources again on the second machine as the second machine will not have a state file. This means that the second run will most likely fail as most resources cannot exist more than once or if it does succeed you will end up with 2 of each resource.

To get around this issue you need to store your state in a remote location. Terraform has different built in storage backends for state. These built in remote backends include AWS S3, AzureRM and Consul plus many more. Consult the Hashicorp Terraform website for a full list. Lets walk through an example of setting up a simple project and use AWS S3 to store our state rather than the default `terraform.tfstate` file.

Start the example by logging in to the AWS console and create an S3 bucket, you can use any name you want, I have used `kevholditch-terraform-state`. You can click through and create the bucket with the default permissions as by default the bucket will only be accessible privately. Next create a new folder and a file called `main.tf` (or if you are using the examples repository then you can find the code in the folder `state_example_03`):

```
provider "aws" {
  region  = "eu-west-1"
  version = "~> 2.27"
}

resource "aws_sqs_queue" "my_queue" {
  name = "my_queue"
}
```

There is not much to the code we are simply setting up the AWS provider and creating a queue. To specify Terraform to use the S3 state backend we need to create another file called `state.tf` in that file paste the following:

```
terraform {
  backend "s3" {
    bucket = "kevholditch-terraform-state"
    key    = "myproject.state"
    region = "eu-west-1"
  }
}
```

This block is configuring the state backend. The `s3` in quotes after the word `backend` tells Terraform that we want to use the AWS S3 backend instead of the default local file backend that we have been using up until now. The bucket parameter is the name of the bucket we want to use. You will need to change this to be the bucket that you created. Next is the key we want to use which essentially is what Terraform will call the state file in S3. Then lastly is the region where we created the S3 bucket.

Go into the folder on the command line and run `terraform init`. You should notice this message:

```
Initializing the backend...

Successfully configured the backend "s3"! Terraform will automatically
use this backend unless the backend configuration changes.
```

Terraform tells you this time that you have configured the backend s3. This means that the state will be stored in the AWS S3 bucket and not in your local file. Go ahead and run `terraform apply` to create the AWS sqs queue. Log into the AWS console and browse to the S3 bucket and you will see that there will be a file there called `myproject.state`. This was the value we gave Terraform for the `key` parameter in the `state.tf` file.

By configuring the project in this way if we wanted to collaborate on a project with multiple people then this would now work ok. We could check this project into source control and anyone working on the project would just have to have access to the AWS S3 bucket where the state is stored.

If you want to work on a project with multiple people then you need to make sure that the state remote backend that you choose supports locking (as the AWS S3 state backend does). When the remote state backend supports locking Terraform will lock the state when it is running preventing anyone else from changing the state at the same time. You will not see a message that Terraform is locking the state, Terraform will just silently do this automatically for you. This is an important feature to have as if the state is not locked then multiple people could overwrite it and it could become corrupted. Another tip is that if you are using AWS S3 then turn on bucket versioning. This will give you every version of your state file. Although you should avoid manually changing the state file unless you are in an extreme situation, having bucket versioning does give you an extra safety net.

One last note before we end this chapter. We put the remote backend state configuration in a file called `state.tf`. This is just a convention used by the community. Terraform actually does not care where you put that block of code, you can put it in any file ending `.tf` including inside the `main.tf`. It is normal practice however to use a file called `state.tf` so it is easy to find the remote state configuration.

Chapter 15 - Workspaces

Workspaces

A workspace in Terraform is a way of creating many instances of a set of Terraform code using a single project. One of the advantages of Terraform is that you can use the infrastructure as code to create your environments reducing human error and making them all the same. You may have wondered up until now how you create multiple environments using the same code as everything we have done up until this point has been around creating a single instance of the infrastructure using our Terraform project. If you ran the project again then it would not create another copy of the infrastructure it would simply say no changes to make.

Workspaces are Terraform's solution to this problem. Let's dive straight into an example then we can go into some detail of how it is working. Create a new folder, and a file called `main.tf` and paste in the following code (or if you are using the examples repository you can find the code inside the folder `workspaces_example_01`):

```
provider "aws" {
  region  = "eu-west-1"
  version = "~> 2.27"
}

resource "aws_sqs_queue" "queue" {
  name = "${terraform.workspace}-queue"
}
```

Most of this code should look familiar to you. The one part you probably will not have seen before is how we are setting the queue name to `"${terraform.workspace}-queue"`. The special variable `terraform.workspace` will assume the value of the current workspace we are running in. All Terraform projects run in a workspace, up until now we have been running in the `default` workspace. The `default` workspace cannot be deleted and is the workspace you will be working in unless you specify otherwise.

Apply the project up to AWS by opening a terminal, changing directory into the folder and running `terraform init` followed by `terraform apply`. Then confirm the apply with `yes` when prompted to do so. Terraform will have created a queue in AWS with the name `default-queue`. Due to the fact that we are working in the `default` workspace.

To list all available workspaces in the current project run the command `terraform workspace list`. This will produce the following output:

```
 terraform workspace list
* default
```

The asterisk marks the workspace we are currently working in. Now let's create a new workspace for our `dev` environment, run the command `terraform workspace new dev`. The new workspace command takes the name of the new workspace as an argument. The output will show that Terraform has created a new workspace and automatically switched to it. Run the list command again, and the following output will now be displayed:

```
 terraform workspace list
  default
* dev
```

The asterisk has now moved to the newly created `dev` workspace. Run `terraform apply` again to apply this Terraform project. Terraform has gone and created a second queue, this one named `dev-queue`. This happened because we have changed the workspace that we are in. Now we are in the `dev` workspace, so all the Terraform commands we run to apply the project or manipulate state and so on will only be applied to all the resources that are part of the `dev` workspace. The `default` workspace where the `default-queue` lives is completely separate. You can think of this like two Terraform projects in two separate folders that happen to be sharing the same code. You can use this functionality to create as many instances of your infrastructure as you wish.

To switch back to the `default` workspace, use the command `terraform workspace select default`. If you make a change to the project now such as changing one of the queue properties and running `terraform apply` that change will be made by Terraform to the `default-queue`. If you go to the AWS console and check the `dev-queue` that will remain unchanged as the apply will have only affected the `default-queue`.

Under the covers the workspace switching works by Terraform keeping multiple copies of the state. Depending on the state backend you are using the technique for how these are stored varies slightly and if you want to take advantage of using workspaces you need to pick a state backend that supports them such as AWS S3.

As we have been using the default local state in this example Terraform simply stores the different workspaces in different local paths on disk. The `default` workspace is stored in the file in the top-level folder called `terraform.tfstate` as we learnt in the previous chapter. Take a look at the state file, and you will see the representation of the `default-queue` in there. The `dev` workspace is in a file at the path `./terraform.tfstate.d/dev/terraform.tfstate`. The special folder `terraform.tfstate.d` is where Terraform stores its local workspaces, each workspace has a folder with its name and in that folder is the `terraform.tfstate` file for that workspace. Under the bonnet Terraform uses a special marker file at the path `.terraform/environment` to store which workspace you are currently using. *Note do not rely on this fact as it is an internal detail and subject to change, I'm detailing how it works to improve your understanding of what is going on.*

Switch back to the `dev` workspace by running `terraform workspace select dev` and destroy the infrastructure (to save the pennies) by running `terraform destroy` and confirming at the prompt.

Switch back to the `default` workspace (`terraform workspace select default`). Let's now delete the `dev` workspace as we are done with it. To delete a workspace use the command `terraform workspace delete dev`. It is important to note that the delete command will remove the Terraform state it will NOT destroy the infrastructure. Which is why we ran the destroy command first!

With the `dev` workspace deleted it will no longer appear in our list of workspaces when we run `terraform workspace list`. Do not forget to destroy the infrastructure in the `default` workspace when you are finished. But remember there is no way to delete the `default` workspace.

Workspaces are an important feature of Terraform that can be used to create several copies of your infrastructure. You can use the workspace name to drive different values for variables that make up your infrastructure. However, the amount that you can drive just from the workspace alone does not scale well. You really need to have a way to change your input variables at the top-level. This problem is solved by Terraform's managed offering Terraform Cloud or Terraform Enterprise. These solutions allow you to create a workspace, point it at a source control repo containing your Terraform code and set the variables for that workspace (you can mark some variables as sensitive, so they cannot be read back). This allows you to have a different set of variables for each workspace. When you get to the level where you want to manage multiple workspaces, I would encourage you to look at one of these offerings as to do this locally requires workarounds with setting and managing variables. Often variables contain sensitive information so this can be hard to do securely.

Chapter 16 - Provisioners

Provisioners

A provisioner in Terraform is a way to run a script either remotely or locally after a resource has been created. They were added by Hashicorp to Terraform to allow for certain scenarios that are not natively supported by the provider you are using. I am including them for completeness, however Hashicorp recommend that they are a last resort. The reason being is that because they are imperative. Therefore, Terraform has no way of knowing how to apply a change you make to your script to the real world, as it does with normal resources.

For example if you have a provisioner script that installs some yum packages on a linux server and then runs a command. If you want to update that provisioner script then you need to make sure that the new updated script works the same way for a machine that has never had the script run or one that has had the old version run. It is this headache that you should be avoiding by using Terraform in the first place which is one of the main reasons they are discouraged.

The example code for provisioners is quite a bit more in depth than the examples we have been using so far. I think that by this point we are ready to take on a bit of a more ambitious Terraform project. To set up this project create a new folder and inside it create a file called `variables.tf` and paste in the following code:

```
variable "my_ip" {}
```

Create a file called `output.tf` and paste in:

```
output "command" {
  value = "curl http://${aws_instance.nginx.public_ip}"
}
```

Lastly, create a file called `main.tf` and paste in the following code (or if you are following along using the examples repo then you can find all of the code in the folder `provisioners_example_01`):

```
provider "aws" {
  region  = "eu-west-1"
  version = "~> 2.27"
}

resource "aws_vpc" "vpc" {
  cidr_block = "10.0.0.0/16"
}

resource "aws_internet_gateway" "main" {
  vpc_id = aws_vpc.vpc.id
}

resource "aws_subnet" "public" {
  vpc_id                  = aws_vpc.vpc.id
  cidr_block              = aws_vpc.vpc.cidr_block
  map_public_ip_on_launch = true
  availability_zone       = "eu-west-1a"
}

resource "aws_route_table" "public" {
  vpc_id = aws_vpc.vpc.id

  route {
    cidr_block = "0.0.0.0/0"
    gateway_id = aws_internet_gateway.main.id
  }
}

resource "aws_route_table_association" "gateway_route" {
  subnet_id      = aws_subnet.public.id
  route_table_id = aws_route_table.public.id
}

resource "aws_security_group" "rules" {
  name   = "example"
  vpc_id = aws_vpc.vpc.id

  ingress {
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["${var.my_ip}/32"]
  }

  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"]
  }
}

resource "aws_key_pair" "keypair" {
  key_name   = "key"
  public_key = file("nginx_key.pub")
}

data "aws_ami" "ami" {
  most_recent = true
  owners      = ["amazon"]

  filter {
    name   = "name"
    values = ["amzn2-ami-hvm-2.0.*-x86_64-gp2"]
  }
}

resource "aws_instance" "nginx" {
  ami                    = data.aws_ami.ami.image_id
  instance_type          = "t2.micro"
  subnet_id              = aws_subnet.public.id
  vpc_security_group_ids = [aws_security_group.rules.id]
  key_name               = aws_key_pair.keypair.key_name


  provisioner "remote-exec" {
    inline = [
      "sudo amazon-linux-extras enable nginx1.12",
      "sudo yum -y install nginx",
      "sudo chmod 777 /usr/share/nginx/html/index.html",
      "echo \"Hello from nginx on AWS\" > /usr/share/nginx/html/index.html",
      "sudo systemctl start nginx",
    ]
  }

  connection {
    host        = aws_instance.nginx.public_ip
    type        = "ssh"
    user        = "ec2-user"
    private_key = file("nginx_key")
  }
}
```

Lets start at the top. The `variable` `my_ip` should be set to your IP address. To find out your current IP address visit `https://www.whatismyip.com/` or if you are a hipster `curl ifconfig.co`. You can then default your IP address as the value of the `my_ip` variable either by setting the `default` parameter or by creating a `terraform.tfvars` file and setting `my_ip = {your ip}` (or any of the other methods we have discussed in the variables chapter). We are using this variable to allow only your IP address through the firewall that we are going to set up in AWS. As it is generally bad practice to open port `22` to the whole world (`0.0.0.0/0`).

Turning to the file `main.tf`. We are setting up a `vpc` which is a virtual private cloud in AWS, essentially this is a ring fenced private network where we are going to put our infrastructure. The internet gateway allows machines inside our VPC to access the internet. Next we create a public subnet and a route table. In the route table we create a default route (`0.0.0.0/0`) and point that to our internet gateway. That is basically saying if you are looking for an IP address outside of the range of our VPC CIDR (`10.0.0.0/16`) then go out through the internet gateway. We then need to associate this route table with our subnet using an `aws_route_table_association`.

By default, AWS security groups which are basically the firewall around a subnet are DENY all. That means to allow traffic in and out of our subnet we need to open up ports in (ingress) and out (egress). This is what we are doing with the `aws_security_group` resource. We open port `22` ingress so that we can ssh onto the server and run a script. We only open up port `22` to your IP (this is what we are using the `my_ip` variable for). We do this because we do not want to open port `22` to the world as it means that anyone can try and get onto that server. We open port `80` ingress so we can hit a website over http running on the server. This is ok to open to any address as its bound to our webserver. Lastly, we create a rule so that all traffic can egress our server on any port. In practice this isn't great and you would want to lock it down, but it makes getting our example up and running easier.

The `aws_key_pair` resource creates a key pair in AWS. A keypair is used to ssh onto a server. The keypair that we are setting up is from a local file in the directory. This keypair **will not exist** on your machine. To create it follow these steps:

1. To generate the keypair run `ssh-keygen`
   File name: `nginx_key`
   No passphrase
2. You should now have two files in this directory `nginx_key` and `nginx_key.pub`.

Note that you should never ever check in the private key from a keypair into source control. This is the file without the `.pub` extension. It is also good practice to set the file permissions on your private key to `400`.

The `aws_ami` block is simply looking up the latest AWS AMI image to use for the EC2 instance that we are going to create.

The last part of the script is the `aws_instance`. We have finally arrived at the resource where our provisioner is defined. A provisioner is essentially a script that Terraform will execute. We start the provisioner block on the resource with the keyword `provisioner`. After `provisioner` the value `remote-exec` is the type of provisioner we want to use, `remote-exec` means that Terraform will execute this provisioner remotely on the resource itself, in this case on the EC2 machine. The other common option is `local-exec` where the script will run on the machine where Terraform is running (ie your machine). There are a few other provisioners available too, such as `chef` and `puppet` but I'm not going to discuss those in this book.

We set the script that we want to run (which is a list of string) as the value of the `inline` parameter. Each element in in the list corresponds to a line in our script. The script installs nginx on our server and starts it on port `80`. For those that do not know nginx is a web server/proxy program, which can do a lot of cool things but for our purposes we are simply using it as a standard web server. The script sets nginx to print the text `Hello from nginx on AWS` when you hit it on port `80`.

The `connection` block tells the provisioner how to connect to the EC2 instance. We set the `host` parameter to the IP address of the server that gets created (`host = aws_instance.nginx.public_ip`), `type` to `ssh` as we want Terraform to ssh onto the box to run the script. The user is `ec2-user`, this is the default user that AWS sets up for use with ssh. We then set the private key to be used to connect via ssh. This is the private part of the keypair that we generated earlier.

Run the project! Change into the directory and run `terraform init` and then `terraform apply`. Terraform will take a couple of minutes to run because it takes a little while to spin up a new instance in AWS. You will see Terraform ssh onto the box and run our script that we defined. We have a handy `output` in our Terraform code which prints out a `curl` command for you to run and test everything is working. When you run this curl you should see `Hello from nginx on AWS`. You can also visit the IP address of the instance you have created in a browser, you should see the same text. Let's take a moment and just absorb what we have just done. We have just created a full private network, setup a firewall, created a webserver, installed a custom script on there and got it to print a message when we visit a page. This should start to show the power of how easy it is to configure infrastructure using Terraform.

It is often so fast to use Terraform that even when I am doing a quick prototype of something I use Terraform. It then has the added advantage that if the prototype works out then you have already written the Terraform code, so you can take that as a first draft to check in and if it does not work out then you can use Terraform to destroy everything.

If you want to ssh onto the machine yourself then you can use the command: `ssh -i nginx ec2-user@{ip}` where `{ip}` is the IP address of the machine that you created. The `-i` denotes to ssh that you want to use a keypair to connect to the machine. Once on the machine you can `curl localhost` to see the same output.

Do not forget to destroy this project when you are done by running `terraform destroy` and confirming with `yes`.

We started this chapter by saying that provisioners should be avoided. If that is the case then you may be wondering if we achieve the same outcome as we have just done by not using a provisioner. The answer is yes! AWS have a built in way to run a script upon new machine start (most cloud providers do). To do this set the script as the `user_data` on the machine and remove the `provisioner` code, here is the changed `aws_instance` to use `userdata` (the code for this is in `provisioners_example_02` folder if you are using the examples repository):

```
resource "aws_instance" "nginx" {
  ami                    = data.aws_ami.ami.image_id
  instance_type          = "t2.micro"
  subnet_id              = aws_subnet.public.id
  vpc_security_group_ids = [aws_security_group.rules.id]
  key_name               = aws_key_pair.keypair.key_name
  user_data = <<EOF
    #!/bin/bash
    set -ex

    yum update -y
    amazon-linux-extras enable nginx1.12
    yum -y install nginx
    chmod 777 /usr/share/nginx/html/index.html
    echo "Hello from nginx on AWS" > /usr/share/nginx/html/index.html
    systemctl start nginx
EOF

}
```

I have altered the script slightly in `user_data`, the reason being that in AWS when `user_data` is being executed it is being run as the root user. Contrast that to when running the script via a provisioner we were running as `ec2-user` which is a limited access user, therefore to make system changes (such as install nginx) we had to elevate to root by prefixing our commands with `sudo`. The other small difference is that we are starting the script with a shebang line (`#!/bin/bash`), this is so that it will be executed with bash and setting `ex` means print each line that is executed and stop upon error. *Note you can see the output of the startup script by examining `/var/log/cloud-init-output.log`*

Null Resources

A `null_resource` is a special no op resource that creates nothing. It is a dummy resource that allows you to attach a provisioner to it. This resource should be avoided unless absolutely necessary.

For example if you want to use a `null_resource` to run the script that sets up the nginx server we have been running that would look like:

```
resource "null_resource" "setup" {
  provisioner "local-exec" {
    command = <<CMD
    ssh -i nginx_key ec2-user@${aws_instance.nginx.public_ip} -o StrictHostKeyChecki\
ng=no -o UserKnownHostsFile=/dev/null 'sudo amazon-linux-extras enable nginx1.12; su\
do yum -y install nginx; sudo chmod 777 /usr/share/nginx/html/index.html; echo \"Hel\
lo from nginx on AWS\" > /usr/share/nginx/html/index.html; sudo systemctl start ngin\
x;'
CMD
  }
}
```

The fact is that to get this to work you would have to put a delay into the script as otherwise it will try and run the command before the server is ready to accept SSH connections. You would never use a `null_resource` for this, as we have explained you would use `user_data` to configure an instance upon startup.

I wanted to include the `null_resource` for completeness, but the fact that I could not come up with a good example of when to use it shows how rarely they are useful in the real world.