IAN MIELL

LEARN TERRAFORM THE HARD WAY

MASTER TERRAFORM USING THE ONLY METHOD THAT WORKS

Learn Terraform The Hard Way

Ian Miell

This book is for sale at http://leanpub.com/learnterraformthehardway

This version was published on 2021-06-01

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.

Also By Ian Miell

Learn Bash the Hard Way

Learn Git The Hard Way

Contents

Learn Terraform the Hard Way

This Terraform course has been written to help users to get to a deeper understanding and proficiency in Terraform. It doesn't aim to make you an expert immediately, but you will be more confident about using it and building your knowledge up from that secure base.

You may well have already played with Terraform a little - maybe inherited some code at work, or hacked something together quickly that does the job. While that's a great way to get going, a guided course that shows you how the pieces fit together by using it really helps you go further. Understanding these pieces enable you to more creative, solving your own challenges in ways that work for your problem domain.

Introduction

Why 'The Hard Way'?

The 'Hard Way' is a method that emphasises the process required to learn anything. You don't learn to ride a bike by reading about it, and you don't learn to cook by reading recipes. Books can help (this one hopefully does) but it's up to you to do the work.

This book shows you the path in small digestible pieces based on my decades of experience and tells you to **actually type out the code**. This is as important as riding a bike is to learning to ride a bike. Without the brain and the body working together, the knowledge does not properly seep in.

If you follow this course, you will get an understanding of Terraform that can form the basis of mastery as you use it going forward.

What You Will Get

This course aims to give students:

- A hands-on, quick and practical understanding of Terraform
- Enough information to understand what is going on as they go deeper into Terraform
- A familiarity with advanced Terraform usage

It does not:

- Give you a mastery of all Terraform's internals

- Give a complete theoretical understanding of all the subtleties and underpinning technologies of Terraform
- Explain everything. There is plenty of time to go deeper and get all the nuances later if you need to. This book will also give you practice in doing your own research to solve your particular challenges

You are going to have to think sometimes to understand what is happening. This is The Hard Way, and it's the only way to really learn. This course will save you time as you scratch your head later wondering what something means, or why that StackOverflow answer worked. You will also be able to construct your own solution and explain why that StackOverflow answer might not be perfect…

Assumptions

This book assumes some familiarity with *very* basic shell usage and commands. For those looking to get to that point, I recommend following this set of mini-tutorials:

https://learnpythonthehardway.org/book/appendixa.html

It also assumes you are equipped with a bash shell and a terminal. If you're unsure whether you're in bash, type:

```
echo $BASH_VERSION
```

into your terminal. If you get a bash version string output like this then you are in bash:

```
3.2.57(1)-release
```

If you are not in bash, then that's not necessarily a problem, but be aware of it as a possible issue if you stumble (and let me know!).

I also assume you have a relatively up-to-date version of Terraform. The Terraform version on my Mac as I write this is:

```
Terraform v0.15.1
```

If you don't have Terraform, then the best place to download it at the time of writing is at:

https://www.terraform.io/downloads.html

You can compare this to yours by typing this on the command line:

```
terraform version
```

How The Course Works

The course *demands* that you type out all the exercises to follow it.

Frequently, the output will not be shown in the text, or even described.

Any explanatory text will assume you typed it out. Again, this is The Hard Way, and we use it because it works.

This is really important: you must get used to working in Terraform, and figuring out what's going on by scratching your head and trying to work it out before I explain it to you. Eventually you will be on our own out there and will need to think for yourself.

Each section is self-contained, and must be followed in full to be sure that it makes sense. To help show you where you are, the shell command lines are numbered 1-n and the number is followed by a $ sign, eg:

```
1 $ first command
2 $ second command
```

At the end of each section is a set of 'cleanup' commands (where needed) if you want to use them.

Part I - Local Basics

This part goes over the very basic concepts that you use when Terraforming.

In it we cover:

- Resources
- Planning
- State files
- Data sources
- Ouptut
- Destruction

You will be using only local resources (such as local files on the host you run Terraform on) so that these concepts get fully embdedded. It won't necessarily look to you like the Terraform you might use at work, but it's fundamenally the same and will give you a good grounding to go further.

Terraform 101

This first section will take you through one of the simplest possible Terraform scripts.

It will create a file in the folder you run it in, and is completely safe.

You will cover the following concepts:

- Terraform resources
- The `path.module` variable
- The `terraform init` command
- The `terraform apply` command

We'll also briefly discuss Terraform's relationship to traditional configuration management tools like Chef, Puppet, and Ansible. By the end you should begin to have a much clearer idea of what Terraform does, and how.

How Important is this Section?

This section is essential, as it introduces core Terraform concepts and commands.

Your First Terraform Module

```
$ mkdir -p ltthw_resources
$ cd ltthw_resources
```

You've created and moved into a folder where you can safely do some work with Terraform without fear of breaking anything else. The other chapters in this book will follow the same pattern.

Now you're going to create a file called `hello_local_file.tf`. If the following commands don't work for you, check you've got the pre-requisites listed at the start of the book.

If you don't understand it, don't worry: just type it in. It's an unambiguous method to edit a file in a bash shell without using an external editor.

```
$ cat > hello_local_file.tf << 'EOF'
> resource "local_file" "hello_local_file" {
>     content     = "Hello terraform local!"
>     filename = "${path.module}/hello_local_file.txt"
> }
> EOF
```

Stop here and think about what you expect that file to do when Terraform 'runs' it. Have a guess as to what a `resource` is, and what `${path.module}` does.

If you're not sure, do a Google search and see what turns up. Don't worry about finding the answer and being right, just get a feel for what looking things up in Terraform looks like. You might find resources you want to look at later.

Done that? OK, now type:

```
9 $ ls -1a
```

You should see something like this:

```
.
..
hello_local_file.tf
```

The `.` represents the current directory, and the `..` represents the parent directory. These are virtual files put in every folder in the filesystem so you can always refer to the parent folder or the current one. The other file is the one just created.

We're checking what's in the folder to show what happens when we bring Terraform in:

> **Note**
>
> You need an internet connection at this point, or you'll get an error about connection failure.

```
10 $ terraform init
```

You'll see some output as Terraform initialises the environment. If you run `ls -1a` again:

```
11 $ ls -1a
```

You'll see an extra `.terraform` folder.

```
.
..
.terraform
hello_local_file.tf
```

Have a root around that folder if you're so inclined. Terraform has downloaded the `local_file` plugin and placed it in this folder.

Now we're ready to 'run' this module, and create the local file:

```
12 $ terraform apply
```

Now (and whenever you run this command and there's something to change), you will get a prompt asking you if you are sure you want to go ahead.

You will need to type `yes` to approve.

Before you do that, read the output carefully. It explains a phase called 'plan' where Terraform takes your module and compares its desired state with what it thinks is the current state in the 'real' world. We will cover plans later. Finally, you get a summary of the plan:

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

Because there's something to change, you get the prompt asking you if you are sure you want to go ahead and make a change. Type `yes` to continue:

```
13 Enter a value: yes
```

You should have seen the output at the end which tells you what was changed:

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

Again, read the output carefully before running `ls -1a` again:

```
14 $ ls -1a
```

```
.
..
.terraform
hello_local_file.txt
hello_local_file.tf
terraform.tfstate
```

Now you see that there's two new files in the folder: `terraform.tfstate` and the `hello_local_file.txt` we created. We will cover the `terraform.tfstate` file later.

Now remove the `hello_local_file.txt` file and re-run `terraform apply`.

What do you think will happen?

```
$ rm hello_local_file.txt
$ ls -1a
$ terraform apply
$ ls -1a
```

Did it do what you expected? What if you change the file?

```
$ echo 'I just moved in' >> hello_local_file.txt
$ cat hello_local_file.txt
$ terraform apply
$ cat hello_local_file.txt
```

What happened?

Terraform vs 'Traditional' Configuration Management Tools

If you've used configuration management tools like Chef, Puppet, or Ansible before, then you may notice that Terraform can do similar things. It can examine the state of something and bring its state into line with the state that was specified, as you saw above.

There are key differences between Terraform and other tools, and these spring from Terraform being designed for infrastructure rather than software configuration.

When managing files (as we saw here), this looks pretty similar to configuration management tools. As we move to managing things such as load balancers, virtual private networks, or fleets of machines, the design decisions of Terraform make it behave in different ways to those tools.

For example, configuration management tools are usually agent-based (ie runs on the machine it's managing), or ssh-based (ie goes to the machine it's managing and does work on it). In other words, they are managing from 'within' the system they are affecting. Terraform, by contrast, must manage 'from outside', as (normally) the management takes place via a remote API.

Fundamentally, though, it's worth keeping in mind that Terraform is a configuration management tool, but designed as one that is for *what your software runs on*, rather than for *what software runs where*.

What You Learned

You've just written and run your first Terraform module.

It might not have been what you expected, in that you affected a local file with your module rather than some cloud infrastructure, like an AWS EC2 instance. This is deliberate: the point of this book is to show you how Terraform works in simple ways so that you grasp the key concepts as quickly as possible. Then you can apply the same concepts to your infrastructure.

Using examples like this, you will gain the knowledge needed to write, debug, explain and maintain Terraform recipes in the real world.

Later you will create and destroy infrastructure items on AWS, but only when that helps show up some key point about Terraform itself.

In this section you learned about:

- Terraform resources
- What the `terraform init` command does
- The files `terraform init` creates
- What the `terraform apply` command does

Cleanup

To clean up what you just did, run:

```
23 $ terraform destroy -auto-approve
24 $ cd ..
25 $ rm -rf ltthw_resources
```

What Next?

In the next section you look more closely at Terraform's state file, which keeps track of the resources the module is concerned with.

Exercises

1) Read the Terraform documentation on the `init` and `apply` commands.

2) Research some of the Terraform resources and what they manage.

Part II - Remote Terraform

In this section you will use Terraform to create resources external to your local host machine. Building on your knowledge from Part I, you'll see how to create resources on a given provider. That might be a cloud-based provider (we'll be using AWS in our examples) or any other kind of resource accessible via an API that's supported by Terraform.

In this part you will cover:

- Terraform providers
- Implicit and explicit resource dependencies
- Graphing Terraform module dependencies
- Importing pre-existing Terraform resources

This will complete the 'basics' part of the course, before you move on to Part III, where you start to cover more advanced Terraform concepts.

Managing Remote Resources

In this first section of Part II you're going to create your first piece of cloud infrastructure using Terraform. I've chosen AWS for my examples as that's the most popular cloud platform today.

This won't cost you a penny (nor will any other section in this book), as you are going to create resources on AWS that are (as of writing) completely free to create (even if you're not on the AWS 'Free Tier'). If this changes, let the author know as soon as possible!

In this section you're going to cover:

- Remote resources
- Providers
- Refreshing resources

How Important is this Section?

Terraform was created specifically to manage remote resources, so this section is essential.

Creating A Remote VPC

AWS Accounts

If you don't already have an AWS account, then you'll need to create one now to follow along. You'll also need to create a key so you can access AWS and create resources using the API. Specifically, you will need an 'access key' and a 'secret key'.

These keys can be created on the AWS web console. See here for more information:

https://docs.aws.amazon.com/general/latest/gr/aws-sec-cred-types.html

The simplest thing to do is create a temporary key that grants 'access to all' or 'admin' access. If you're not comfortable creating a temporary admin key, then you will need to create an IAM role that gives you the capability to create all the resources we want to here.

First, type this out and think about what's new as you're doing it. As you come across something you don't understand you might want to quickly research it (but don't worry if your researches don't make too much sense, as we cover the ground as we go):

Secret Key Management

In the below listing, and all future 'aws' ones, you'll need to replace `YOURACCESSSKEY` and `YOURSECRETKEY` with the secret key values you covered in the 'AWS Accounts' note above.

> You may also want to keep the upcoming `provider` section handy on your machine as you follow the book so you don't have to keep typing it out by hand.

```
$ export AWS_SECRET_ACCESS_KEY=YOUR_AWS_SECRET_ACCESS_KEY
$ export AWS_ACCESS_KEY_ID=YOUR_AWS_ACCESS_KEY_ID
```

```
$ mkdir -p ltthw_aws_vpc
$ cd ltthw_aws_vpc
$ cat > hello_remote.tf << EOF
> provider "aws" {
>   region     = "us-east-1"
> }
> resource "aws_vpc" "main" {
>   cidr_block = "10.0.0.0/16"
> }
> EOF
```

You should have noticed that you created another Terraform module similar to previous ones. This time, in addition to the usual `resource` stanza, you created a `provider` stanza. Also, we introduced a new resource type: `aws_vpc`.

`aws_vpc` is an example of a 'remote resource'. It's a resource managed remotely instead of locally on your machine. In this case, `aws_vpc` is a 'virtual private cloud' on AWS. This provides you a network space that you have control over, and in which you can add other AWS resources (such as virtual machines with AWS's EC2 service, or databases with AWS's RDS service). The `cidr_block` allocates a set of private IP addresses to this VPC.

> ## Region
>
> I've used the generally-considered default region `us-east-1` above. You can change this to your preferred region if you have one, but remember to do that everywhere regions are mentioned in the remainder of the book.

A provider is the part of Terraform that interacts with any external APIs to make resources available to you. There are many mainstream cloud providers including:

- AWS
- Azure

- OpenStack

In addition, there are many providers you might *not* expect to see, such as:

- GitHub
- Kubernetes
- MySQL
- Mailgun

These providers provision resources on those services. For example, the GitHub provider allows you to add membership of users to teams, configure branches, set up teams, and many other resources on GitHub.

Again, you may want to spend a little time researching these providers and what they do at this point, especially if you have a use case in mind for Terraform.

Now run the Terraform module to get your VPC created:

```
$ terraform init
$ terraform apply
```

Running `terraform apply` will give you output describing the resource creation that looks 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_vpc.main will be created
  + resource "aws_vpc" "main" {
      + arn                              = (known after apply)
      + assign_generated_ipv6_cidr_block = false
      + cidr_block                       = "10.0.0.0/16"
      + default_network_acl_id           = (known after apply)
      + default_route_table_id           = (known after apply)
      + default_security_group_id        = (known after apply)
      + dhcp_options_id                  = (known after apply)
      + enable_classiclink               = (known after apply)
      + enable_classiclink_dns_support   = (known after apply)
      + enable_dns_hostnames             = (known after apply)
      + enable_dns_support               = true
      + id                               = (known after apply)
```

```
      + instance_tenancy                = "default"
      + ipv6_association_id             = (known after apply)
      + ipv6_cidr_block                 = (known after apply)
      + main_route_table_id             = (known after apply)
      + owner_id                        = (known after apply)
    }

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

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

  Enter a value: yes

aws_vpc.main: Creating...
aws_vpc.main: Still creating... [10s elapsed]
aws_vpc.main: Creation complete after 13s [id=vpc-07053dc65c34e248f]
```

The '`(known after apply)`' items are a useful placeholder that indicate that the value will be given to you by the provider of the resource once the resource is created. `cidr_block` was given by us, and the Terraform provider code given to us for AWS sets a default for `assign_generated_ipv6_cidr_block` and `enable_dns_support` of `false` and `true` respectively.

At this point, you can look at what Terraform thinks the state is with a `terraform show` command:

```
15 $ terraform show terraform.tfstate


# aws_vpc.main:
resource "aws_vpc" "main" {
    arn = "arn:aws:ec2:us-east-1:701780912049:vpc/vpc-07053dc65c34e248f"
    assign_generated_ipv6_cidr_block = false
    cidr_block                       = "10.0.0.0/16"
    default_network_acl_id           = "acl-0190de7297f62c4ae"
    default_route_table_id           = "rtb-0f6085dd75274c332"
    default_security_group_id        = "sg-01822fa63ddffc7ff"
    dhcp_options_id                  = "dopt-5d671426"
    enable_classiclink               = false
    enable_classiclink_dns_support   = false
    enable_dns_hostnames             = false
    enable_dns_support               = true
    id                               = "vpc-07053dc65c34e248f"
```

```
    instance_tenancy                = "default"
    main_route_table_id             = "rtb-0f6085dd75274c332"
    owner_id                        = "701780912049"
}
```

Notice how all the items previously marked as `(known after apply)` are now filled out.

Changing A Resource Outside Terraform

Now you're going to change the resource you just created without using Terraform. To do this you'll need to log into your AWS web console, and find the just-created VPC resource.

What is a VPC?

VPC stands for 'Virtual Private Cloud'. This is a virtual network dedicated to your AWS account.

Make sure you are in the correct region (which will be `us-east-1`, unless you changed it above). If you can't find it, you may want to try visiting

https://console.aws.amazon.com/vpc/

and picking the VPC with the ID you saw in the `terraform show` output.

Once you've found your VPC, give it a name by hand in the console.

To do this, go to the 'Tags' tab at the bottom of the VPC page and add a tag called 'Name' with a name of your choice (eg `ltthwvpc`).

Now you are in a state where Terraform has one view of the resource's state (it has no name as far as Terraform is concerned), while the *provider* of that resource (AWS) has another (the name you just gave it in the console).

Terraform is therefore in an *inconsistent state* with the resource. We saw with local files before that this was a problem for Terraform. However, there is a command that can help:

```
16 $ terraform refresh
```

Read the output carefully. What did that do?

Now if you look again at your state file, you will see it has the name you gave it:

```
17 $ terraform show terraform.tfstate
```

Finally, you're going to destroy the resource you just created.

```
$ terraform destroy -auto-approve
```

Now go back to the AWS console to confirm that it has gone.

Cleanup

To clean up what you just did, run:

```
$ cd ..
$ rm -rf ltthw_aws_vpc
```

What You Learned

In this section you created your first cloud provider resource using Terraform, updated it by hand, and then used `terraform refresh` to make the state consistent again.

What Next?

In the next section you're going to look at how Terraform manages dependencies for you, and how you can also specify dependencies for it to manage for you.

Exercises

1) Go back to Part I and use 'refresh' against the files created to embed the knowledge of how the state file and refreshing works.

2) Go through all the providers documented on the Terraform docs and find out what each one does. This will give you a good idea of the extent of Terraform's current capabilities.

Dependencies

In this section you're going to get to grips with dependencies in Terraform. One of Terraform's key selling points is that it can manage dependencies between different parts of your infrastructure.

You'll cover how Terraform:

- Can create implicit dependencies for you
- Can be used to define explicit dependencies
- Manages targeted destruction of resources

These dependencies are the bane of any infrastructure manager's life. If you switch off that VM who will complain? What about that Load Balancer?

Stories of 'just switching off servers to see who complains' as a way of managing unknown dependencies are incredibly common in the IT industry. Terraform can help manage that problem by codifying and tracking resource dependencies, but it's important to remember that it does not do so by magic, and that your application dependencies may still need to be mapped out by whoever's in charge of them.

How Important is this Section?

Terraform scripts that are in any way sophisticated will use dependencies, both explicit and implicit, so this section is important if you plan to use scripts that go beyond the basics.

Implicit Dependencies

First, set up your credentials as you did the previous section:

```
$ export AWS_SECRET_ACCESS_KEY=YOUR_AWS_SECRET_ACCESS_KEY
$ export AWS_ACCESS_KEY_ID=YOUR_AWS_ACCESS_KEY_ID
```

Create another folder with a new Terraform module:

```
$ mkdir -p ltthw_dependencies
$ cd ltthw_dependencies
$ cat > dependencies.tf << EOF
> provider "aws" {
>   region     = "us-east-1"
> }
> EOF
```

So far, so familiar. You've created a Terraform module with an AWS provider defined.

Now you're going to add two resources to that file. The first is an AWS VPC, similar to the one you created before.

This time you're going to give it a name directly in the Terraform module (`ltthw-vpc`) rather than doing so manually in the AWS console:

```
$ cat >> dependencies.tf << EOF
> resource "aws_vpc" "ltthw-vpc" {
>   cidr_block = "10.0.0.0/16"
>   tags = {
>     Name = "ltthw-vpc"
>   }
> }
> EOF
```

Now the second resource, which is of the type `aws_subnet`. This subnet is attached to the VPC, and is linked by the VPC id.

That raises a problem: before you create the VPC, you don't know its ID (it's given to you by AWS on creation). Yet you want to refer to it so that you can link the subnet to the VPC via the `vpc_id`.

What is a subnet?

A subnet is a range of IP addresses that you can route network traffic to or set other rules against. If you don't understand this, don't worry about it for this book, but if you're going to manage infrastructure on cloud providers you may want to up your networking knowledge.

```
$ cat >> dependencies.tf << 'EOF'
> resource "aws_subnet" "ltthw-vpc-subnet" {
>   vpc_id = aws_vpc.ltthw-vpc.id
>   cidr_block = aws_vpc.ltthw-vpc.cidr_block
> }
> EOF
```

In Terraform, every resource has attributes that you can reference with the syntax `TYPE.NAME.ATTRIBUTE`. Remembering the three-letter acronym 'TNA' might help you remember this pattern of referencing.

> ## AWS Permissions
>
> We assume here that your AWS credentials have permissions to create an AWS VPC. If you run into problems it may be because your account is limited and you'll need to expand its permissions, or use a different one.

```
$ terraform init
$ terraform plan
$ terraform apply -auto-approve
```

Next you're going to run a `terraform plan -destroy` command with an extra flag to specify that you only want to destroy the VPC.

```
$ terraform plan -destroy -target=aws_vpc.ltthw-vpc
```

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

Read the full output on your terminal carefully.

You didn't destroy anything there, but you did see that the subnet and the VPC were *both* planned for destruction. This is despite the fact that you specified `-target=aws_vpc.ltthw-vpc`.

This is because there is an implicit dependency between the subnet and the VPC. This dependency is understood and managed by Terraform for you. It's part of the logic of the Terraform AWS provider.

If you were to run `terraform apply` at this point, it would delete both items. To cancel the effect of this plan to destroy, rerun a plan without the `-destroy` flag. This puts the terraform plan back to a state where nothing needs changing.

```
$ terraform plan
```

Try the destruction above again, but this time use the flag: `-target=aws_vpc.ltthw-vpc-subnet` to target *only* the subnet, not the vpc:

```
$ terraform plan -destroy -target=aws_subnet.ltthw-vpc-subnet
```

Now you can see that this time *only* the subnet is marked for destruction. It was targeted, and no other resource was dependent on it.

Now destroy everything you have created so far:

```
$ terraform destroy -auto-approve
```

Order is Unimportant

Now you're going to do the same thing, but this time you're going to reverse the order of declaration. The subnet will be declared first, and then the VPC. What do you think will happen?

```
$ cat > dependencies.tf << 'EOF'
> provider "aws" {
>   access_key = "YOURACCESSKEY"
>   secret_key = "YOURSECRETKEY"
>   region     = "us-east-1"
> }
> resource "aws_subnet" "ltthw-vpc-subnet" {
>   vpc_id = aws_vpc.ltthw-vpc.id
>   cidr_block = aws_vpc.ltthw-vpc.cidr_block
> }
> resource "aws_vpc" "ltthw-vpc" {
>   cidr_block = "10.0.0.0/16"
>   tags = {
>     Name = "ltthw-vpc"
>   }
> }
> EOF
$ terraform init
$ terraform plan
$ terraform apply -auto-approve
```

What happened? Was it what you expected?

This should show you that *the ordering of resources within a module is unimportant.* Terraform takes the resources declared and creates a dependency graph that allows it to know what needs to be done on each run. In the next section you'll look at how to visualize this graph.

If you've written a query in SQL that queries a database, Terraform works a bit like that: with a SQL query you specify the data you want returned and its constraints. How it gets that data out from the database and returned to you is up to the query parser, optimiser and database management system.

Or, if you've compiled a program you might already know that a program is translated to machine code, which might itself process things in a different order to the order you told it to run in. Even the chip the code runs on might order its internal instructions differently to the way you declared it.

The point is that Terraform allows you to specify the infrastructure you want, and it takes care of the detail of what needs to happen for that to be provisioned, updated, or destroyed. Dependencies are one aspect to this; '`(known after apply)`' data, and other defaults, are another.

Destroy everything again:

```
$ terraform destroy -auto-approve
```

Explicit Dependencies

You just created a resource that Terraform knew was dependent on another (the VPC subnet). It knew it was dependent because the logic within the provider code 'knows' that.

But what about dependencies that are *not* part of the structure of resources themselves, but part of the logic of *your* application?

For example, you might have a dependency in your application between an EC2 virtual machine on AWS and a load balancer on Azure. This would not be captured automatically by Terraform, but fortunately you can specify it.

Type this scenario out and try and figure out what's different this time:

```
$ cat > dependencies.tf << 'EOF'
> provider "aws" {
>   access_key = "YOURACCESSKEY"
>   secret_key = "YOURSECRETKEY"
>   region     = "us-east-1"
> }
> resource "aws_vpc" "ltthw-vpc" {
>   cidr_block = "10.0.0.0/16"
>   tags = {
>     Name = "ltthw-vpc"
>   }
> }
> resource "local_file" "hello_local_file" {
>     content    = "Hello terraform local!"
>     filename   = "${path.module}/hello_local.txt"
>     depends_on = [aws_vpc.ltthw-vpc]
> }
```

```
> EOF
$ terraform init
$ terraform plan
$ terraform apply -auto-approve
```

Now if you plan to destroy the VPC, what do you think will happen?

```
$ terraform plan -destroy -target=aws_vpc.ltthw-vpc
```

Did that do what you expected? Can you explain why both were planned for destruction?

What about this one? Try and predict what it will plan to do:

```
$ terraform plan -destroy -target=local_file.hello_local_file
```

Did you get it right? If so, you now know how to set explicit dependencies.

Now destroy what you created:

```
$ terraform destroy -auto-approve
```

In recent versions of Terraform, it warns you that `-target` is not for routine use, and should only be used in when recovering from a problem, or when directed to by Terraform in an error message.

What You Learned

- The `-target` flag to `terraform plan` and `destroy`
- The difference between an implicit and explicit dependency
- How to specify an explicit dependency

Cleanup

To clean up what you just did, run:

```
$ cd ..
$ rm -rf
```

What Next?

In the next section you're going to look at how to visualise your terraform resources and their dependencies in graph form.

Exercises

1) Go back to previous section and try using 'refresh' against the files after changing them to update the Terraform state.