Gorgo Tasting OpenTofu and Terraform

September 08, 2026

Contents

Chapter 0

How to use this booklet

This is a short booklet to give you a taste of OpenTofu. By the end you will have a web server that serves sayings, backed by a database, running first on your machine, then on Kubernetes, and finally on the internet with a real name and a real certificate. All of it is created, changed, and torn down with one command.

Each chapter is meant to help you understand a topic, but you will still want to reference the provider documentation for the specifics of every argument. Hopefully, after this taste, you will have the context you need to dig into deeper reference material.

OpenTofu and Terraform

Terraform was created by HashiCorp in 2014 and became the standard way to describe infrastructure as code. In 2023 HashiCorp changed its license to one that is no longer open source, and the community forked the last open source release as OpenTofu under the Linux Foundation. The two projects share the same language, the same providers, the same state file format, and the same commands. The differences are at the edges, and Appendix A lists the ones that matter.

This booklet treats the two as equivalent. Commands are written as `tofu`; if you use Terraform, type `terraform` instead. Every configuration in this booklet works with both, and where one has a feature the other lacks, the booklet does not use it.

Wut: The top-level configuration block is called `terraform` even in OpenTofu. OpenTofu kept the name so that existing configurations keep working. When you see `terraform { ... }` in a file, read it as "settings for OpenTofu".

Callouts

Tips call out details that you need to pay special attention to. **Traps** warn you of common mistakes. **Wut** calls out a detail that is counter-intuitive, so make sure you pay attention.

What you need

- **OpenTofu** (or Terraform). On macOS, `brew install opentofu`. On Ubuntu, `snap install --classic opentofu`, or follow the installer instructions at opentofu.org. On Windows, `winget install OpenTofu.Tofu`. Check with `tofu version`.
- **Docker** running on your machine (Chapters 2 through 5): Docker Engine on Linux, Docker Desktop on macOS and Windows.

- **Go** 1.21 or newer (Chapters 2 and 3, for running the server outside a container): the installers at go.dev/dl cover all three systems, or `brew install go` on macOS and `winget install GoLang.Go` on Windows. The examples declare Go 1.26 in `go.mod`, and any Go from 1.21 on fetches that toolchain for you the first time it is needed.
- **kind** and **kubectl** (Chapter 4). On macOS, `brew install kind kubectl`. On Windows, `winget install Kubernetes.kind` and then `winget install Kubernetes.kubectl`. On Linux, the release binaries from the two projects' sites.
- For Chapter 5, an **Oracle Cloud** free tier account, a domain whose Domain Name System (DNS) records are hosted by **Cloudflare** (free is fine), and an email address for **Let's Encrypt**.

Windows

The sessions in this booklet use a POSIX (Portable Operating System Interface) shell, which is what macOS and Linux give you. On Windows you have two good options.

The simplest is **WSL** (Windows Subsystem for Linux): install Ubuntu from the Microsoft Store, install Docker Desktop with its WSL integration turned on, and then follow the Linux instructions inside the Ubuntu terminal. Everything in the booklet works there exactly as printed.

The other is to stay in **PowerShell** with the native Windows tools, which works too, with a few translations:

In the booklet	In PowerShell
`export NAME=value`	`$env:NAME = "value"`
`NAME=value tofu apply`	`$env:NAME = "value"; tofu apply`
`$(command)`	the same
`curl`	`curl.exe` (plain `curl` is an alias for a different command)
`command &` (run in the background)	run it in a second terminal
`~/.ssh/id_ed25519.pub`, `~/.kube/config`	the same; OpenTofu expands `~` to your home directory

OpenTofu itself, its language, and the providers behave the same on all three systems, and so do the configurations in this booklet.

Reading the examples

Configuration is shown in `terraform` code blocks:

```
resource "terraform_data" "hello" {
  input = "hello world"
}
```

Terminal sessions are shown in an outlined box. Bold lines that start with $ are what you type — don't type the $; everything else is what the tool printed back:

```
$ tofu plan
Plan: 1 to add, 0 to change, 0 to destroy.
```

Output is trimmed to the interesting lines, and a line with only `...` stands for output that was left out. Your output will have more in it, and the ids and timings will differ.

The complete, tested configurations for every chapter are in the `examples` directory next to this booklet, under the directory names the chapters use: `hello-world`, `webserver`, `database`, `kubernetes`, `oracle`, the shared

`modules`, and `motd`, the sayings module that Chapters 3 to 5 pull from GitHub. Each one is self-contained, so you can `cd` into it and run `tofu init` and `tofu apply` without copying anything else.

Chapter layout

Each chapter follows a consistent structure:

- **The need, why it matters, why it is hard, and the strategy** at the top, so you know the problem the chapter solves before it starts solving it.
- The **main content** with explanations, configuration, commands, and callouts.
- **Key Points** summarizing the concepts that were introduced.
- **New Syntax** collecting the language elements the chapter introduced in one table.
- **Try It** suggestions of things to change, break, and extend.
- **Exercises** with a mix of questions to test your understanding.

New syntax is introduced when a chapter needs it, with an attempt to explain why it exists and how to think about it, not just what it is.

Appendices

Appendix A collects best practices, recommendations, and common errors for OpenTofu and Terraform in one place. Skim it once now so that you know it exists, and read it properly after Chapter 3, when the advice will make sense. The chapters point to it when they run into one of its items.

Appendix B lists what OpenTofu provides without any provider: the built-in resource and data source, the block types, the meta-arguments, the named values, the operators, and the commonly used built-in functions with an example of each. Keep it open while you write configuration.

Exercises

Do not skip the exercises at the end of the chapters. You can get the answer key, but do not look at it before you work out the answer yourself. If you peek first, the concepts will not sink in. And most of all, run the examples. Infrastructure as code is learned by applying, breaking, and destroying things, and there is no other way that comes close to it.

Chapter 1

Hello, World

The need. You have something that must exist: a running program, a server, a database, a DNS record. Today you make it exist by typing commands, clicking through a console, or following a wiki page that was accurate two years ago. Tomorrow you need it again on another machine, or you need to change one thing, or a colleague needs to know what you did.

Why it matters. Infrastructure you cannot reproduce is infrastructure you cannot fix, review, or hand off. Writing it down as code means it can be versioned, diffed, reviewed in a pull request, and rebuilt from nothing after a disaster or a mistake.

Why it is hard. The world is stateful. A script that creates a server works the first time and fails the second time because the server already exists. Steps depend on each other, scripts fail halfway through, and the only record of what exists is whatever you remember.

The strategy. Instead of writing steps, you describe the desired end state in files. OpenTofu compares the files with what exists, shows you the difference as a plan, and applies only the difference. It remembers what it created in a state file so that the next run starts from what is already there. This chapter builds the smallest possible example of that loop: a configuration that runs `echo hello world`.

The smallest configuration

Create the directory `hello-world`, and in it a file named `main.tf`:

```
resource "terraform_data" "hello" {
  provisioner "local-exec" {
    command = "echo hello world"
  }
}
```

That is the whole thing. Before running it, read it the way OpenTofu does.

The language is made of **blocks**. A block has a type (`resource`), zero or more labels in quotes (`"terraform_data"` and `"hello"`), and a body in braces holding **arguments** (`command = "..."`) and nested blocks (`provisioner`). Think of a block as a paragraph that describes one thing.

A `resource` block is a noun: a thing that should exist. Its first label is the resource **type**, which decides what kind of thing it is and which arguments it accepts. `terraform_data` is a built-in resource type that manages nothing at all. It is a placeholder: a thing that exists only so that you can attach behavior to it or hang other things off it. That makes it perfect for a first example, because it needs no accounts, no plugins, and no network. Its second label is the **name**, which is how the rest of the configuration refers to this particular one. Together they form the address `terraform_data.hello`.

A `provisioner` block says "when this resource is created, also do this". `local-exec` runs a shell command on the machine where you run `tofu`. Provisioners are an escape hatch that you will use rarely in real work (see Appendix A), but they let OpenTofu run `echo`, which is exactly what you want right now.

The workflow

Every configuration directory must be initialized with `init`. Initialize `hello-world`:

```
$ tofu init
OpenTofu has been successfully initialized!
```

`init` prepares the working directory: it downloads any plugins the configuration needs and sets up where state is stored. This configuration needs no plugins, so `init` has almost nothing to do, but it is still required. You run it again whenever the plugins or the state settings change; OpenTofu reminds you when you forget.

Next, ask what would happen:

```
$ tofu plan
OpenTofu will perform the following actions:

  # terraform_data.hello will be created
  + resource "terraform_data" "hello" {
      + id = (known after apply)
    }

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

The **plan** is a diff between the files and reality. `+` means create, `-` means destroy, `~` means change in place, and `-/+` means destroy and recreate. `(known after apply)` marks values, like an id, that only exist once the thing is real. The last line is the summary you learn to read first.

Now make it so:

```
$ tofu apply
Plan: 1 to add, 0 to change, 0 to destroy.

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

  Enter a value: yes

terraform_data.hello: Creating...
terraform_data.hello: Provisioning with 'local-exec'...
terraform_data.hello (local-exec): Executing: ["/bin/sh" "-c" "echo hello world"]
terraform_data.hello (local-exec): hello world
terraform_data.hello: Creation complete after 0s [id=3619212a-1cff-2c5e-3400-ff0bbf709872]

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

`apply` plans again, asks for confirmation, and then performs the actions. There is your hello world, printed by the provisioner during creation.

Now run `tofu apply` a second time:

```
$ tofu apply
No changes. Your infrastructure matches the configuration.

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

This is the most important idea in the booklet. You did not tell OpenTofu to run `echo` — you told it that `terraform_data.hello` should exist. It already exists, so there is nothing to do. A configuration describes an end state, and applying it twice is safe.

Finally, take it all back:

```
$ tofu destroy
terraform_data.hello: Destroying... [id=3619212a-1cff-2c5e-3400-ff0bbf709872]
terraform_data.hello: Destruction complete after 0s

Destroy complete! Resources: 1 destroyed.
```

`destroy` plans the removal of everything in the state and asks for confirmation. Nothing is printed this time: provisioners run at creation, not destruction.

Tip: Get into the habit of running `plan` before `apply`, even though `apply` plans for you. Reading the plan is where you catch the surprise `-/+` on a database. When the plan is what you expected, `apply` is boring, which is exactly what you want.

State

Look in the `hello-world` directory after the first `apply`:

```
$ ls
main.tf  terraform.tfstate
```

`terraform.tfstate` is OpenTofu's memory. It records which real things correspond to which resource blocks, along with their ids and attributes. That is how the second `apply` knew there was nothing to do, and how `destroy` knew what to remove.

You can ask what the state contains:

```
$ tofu state list
terraform_data.hello
```

State is a JSON (JavaScript Object Notation) file, so you will be tempted to read it. Reading is fine. Editing it by hand is how you end up with a tool that thinks a server exists when it does not, or misses one that does. Use `tofu state` subcommands when you need to change it (Appendix A lists them).

Trap: State often contains secrets, because passwords, keys, and tokens end up in it as ordinary attributes. Never commit `terraform.tfstate` to git. Add `*.tfstate` and `*.tfstate.*` to `.gitignore` before your first `apply`, not after. Chapter 5 talks about where state should live when more than one person works on it.

Making it configurable

Hard-coding "hello world" in the command is fine for a demo and useless for anything else. Replace `main.tf` with a version that takes the greeting as an input:

```
terraform {
  required_version = ">= 1.6" # OpenTofu 1.6, Terraform 1.6, or newer
}

variable "greeting" {
  type        = string
  description = "What the program prints."
  default     = "hello world" # remove this line to make the greeting required

  validation {
    condition     = length(trimspace(var.greeting)) > 0
    error_message = "greeting must not be blank."
  }
}

# a placeholder resource; running the provisioner is all it does
resource "terraform_data" "hello" {
  input            = var.greeting
  triggers_replace = [var.greeting] # a new greeting means a new resource

  provisioner "local-exec" {
    command = "echo '${self.input}'"
  }
}

output "greeting" {
  description = "What was printed."
  value       = terraform_data.hello.output
}
```

There is a lot of new syntax here, so take it a block at a time. # starts a comment that runs to the end of the line; OpenTofu ignores it, and the next reader is grateful for it. Read the comments in this example to better understand what the code is doing.

The `terraform` block holds settings for OpenTofu itself. `required_version` refuses to run with a tool older than 1.6 — the release where OpenTofu began. A version constraint is a promise about what you tested with. It means a colleague with an old install gets a clear error instead of a strange one.

A `variable` block declares an **input variable**: a value the user of the configuration can set. Think of it as a function parameter. `type` says what kind of value is allowed, `description` explains what the variable is for, and `default` makes it optional. A variable without a default must be supplied, or `apply` will stop and ask for it.

Inside expressions the variable is `var.greeting`. The `var.` prefix is there so that variables cannot be confused with resources.

`"echo '${self.input}'"` is a string with an **interpolation**. Read `${...}` as "paste the value of this expression here". Anything outside the braces is literal text. When the whole value is one expression, you do not need a string at all: `input = var.greeting` refers to the value directly instead of pasting it into text.

`self` is the resource the provisioner belongs to — in this case, a `terraform_data` resource. Two of its fields matter to the provisioner: the `input` argument stores any value you like, and the `output` attribute is set to the same value as `input` after creation. Together they are a convenient way to move a value into the provisioner. You cannot add fields of your own. `terraform_data` has exactly `input`, `triggers_replace`, `output`, and `id`. If you want to pass more values, you could use an object as the input. For example:

```
input = { greeting = "hola", repeat = 3 }
```

An `output` block declares an **output value**: something the configuration reports when it finishes. It is the return value of the function. Outputs are printed at the end of `apply`, and you can read them any time with `tofu output`:

```
$ tofu output
greeting = "hello world"
```

```
$ tofu output -raw greeting
hello world
```

`-raw` prints the value with no quotes, which is what you want when feeding an output to another command.

Setting variables

There are three ways to supply a variable, and you will use all of them:

```
$ tofu apply -var 'greeting=hola mundo'
```

```
$ TF_VAR_greeting='hola mundo' tofu apply
```

Or put values in a file named `terraform.tfvars`, which is loaded automatically:

```
greeting = "hola mundo"
```

The same file can be written as JSON under the name `terraform.tfvars.json`, which is the form to reach for when a script writes the values instead of a person:

```
{
  "greeting": "hola mundo"
}
```

The environment form is how you pass values from scripts and continuous integration (CI) without putting them on a command line where `ps` can see them.

Nothing stops you from using several of these at once, so OpenTofu reads every source it can find and lets the later ones overwrite the earlier ones. From weakest to strongest:

1. the `default` in the `variable` block
2. `TF_VAR_greeting` in the environment
3. `terraform.tfvars`
4. `terraform.tfvars.json`
5. every `*.auto.tfvars` and `*.auto.tfvars.json`, in alphabetical order by file name
6. `-var` and `-var-file` on the command line, in the order you write them

The surprise in that list is the environment, which loses to every file. Set `greeting` in `terraform.tfvars` and `TF_VAR_greeting` is ignored, with nothing printed to say so.

Trap: A value from `-var` or `TF_VAR_` is used for one command and then forgotten. Nothing writes it back to `terraform.tfvars`, and a root module's variables are not kept in the state file either, so the value is gone when the command exits while whatever it built remains. The next plan resolves the variable from the file or the default again and proposes to undo the change. If a value should stick, put it in a file.

Validation

A configuration can be wrong in several ways, and OpenTofu checks for each kind in its own layer. The earlier a layer catches a problem, the cheaper it is.

Layer 1: formatting. `tofu fmt` rewrites your files in the canonical style: two-space indentation, aligned `=` signs, no trailing spaces. Run it before every commit, or let your editor run it on save. CI runs `tofu fmt -check`, which exits non-zero if anything would change. `tofu fmt -diff` shows what it would rewrite, without rewriting it:

```
$ tofu fmt -diff
--- old/main.tf
+++ new/main.tf
@@ -1,3 +1,3 @@
 output "greeting" {
-    value="hello"
+  value = "hello"
 }
```

Layer 2: structure. `tofu validate` checks that the syntax is valid, that every referenced thing exists, and that every argument is one the resource type accepts. It needs no credentials and touches nothing, so it is safe to run anywhere. Misspell `command` as `comand` and it tells you:

```
$ tofu validate
Error: Unsupported argument

  on main.tf line 20, in resource "terraform_data" "hello":
  20:     comand = "echo '${self.input}'"

An argument named "comand" is not expected here. Did you mean "command"?
```

Layer 3: values. The `validation` block inside a variable is a rule about the value. `condition` is an expression that must be true, and `error_message` is what the user sees when it is not. Try a blank greeting:

```
$ tofu plan -var 'greeting=   '
Error: Invalid value for variable

  on main.tf line 5:
   5: variable "greeting" {
    ├────────────────
    │ var.greeting is "   "

greeting must not be blank.

This was checked by the validation rule at main.tf:10,3-13.
```

The condition uses two built-in **functions**. `trimspace` strips leading and trailing whitespace and `length` counts characters. Functions are called the way you would expect. There are a lot of them: string, number, collection, date, hash, and file functions. You will see more of them as the booklet goes on, and the full list is in the documentation.

Layer 4: the plan. `tofu plan` is the last check before anything happens. It runs every layer above it, then talks to the real world and tells you what would change. Later chapters add two more validation tools that run during plan and apply: `check` blocks that test the result (Chapter 2), and `precondition` and `postcondition` blocks that make assumptions explicit (Chapter 3).

Changing things

Apply the new configuration, then apply it again with a different greeting:

```
$ tofu apply -var 'greeting=hola mundo'
  # terraform_data.hello must be replaced
-/+ resource "terraform_data" "hello" {
      ~ id               = "3619212a-..." -> (known after apply)
      ~ input            = "hello world" -> "hola mundo"
      ~ output           = "hello world" -> (known after apply)
      ~ triggers_replace = [
          - "hello world",
          + "hola mundo",
        ]
    }

Plan: 1 to add, 0 to change, 1 to destroy.
...
terraform_data.hello (local-exec): hola mundo
```

The plan says the resource "must be replaced", and the changed `triggers_replace` list is why. Some changes can be made in place; others need the old thing destroyed and a new one created. Each resource type decides which is which, and the plan always tells you: `~` for an update in place, `-/+` for a replacement.

`triggers_replace` deserves a closer look, because it is doing the real work in this configuration. It takes a list of values. When the resource is created, OpenTofu records the list in state along with everything else about the resource. On every later plan it evaluates the list again and compares it with what was recorded. If every element is the same, the resource is left alone; if any element differs, the resource is planned for replacement, no matter what else changed. Think of the list as the resource's identity: as long as the identity holds, it is the same thing; when the identity changes, it is a different thing, and has to be created.

Why does `terraform_data` need such an argument when other resource types do not? A container or a database has real attributes, and the provider knows which of them can be changed in place and which cannot; a new image, for instance, means a new container. `terraform_data` manages nothing real, so no part of it ever *needs* replacing. Left alone, OpenTofu would happily update `input` in place forever. `triggers_replace` is how you tell it which values matter. Here the greeting is listed, so a new greeting means a new resource, and because provisioners run at creation, a new resource means another `echo`. That is also why the greeting appears twice in the block: `input` carries it to the provisioner through `self.input`, and `triggers_replace` turns a changed greeting into a replacement.

The list can hold more than one value, and a change to any one of them is enough to replace the resource. When you want a replacement without changing anything, `tofu apply -replace=terraform_data.hello` forces one from the command line. Chapter 3 introduces `replace_triggered_by`, the general form of this idea: it works on any resource type and watches other resources instead of values.

Run the same apply a second time, with the same greeting, and nothing happens:

```
$ tofu apply -var 'greeting=hola mundo'
No changes. Your infrastructure matches the configuration.

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

The trigger list matches what the state recorded, so there is no replacement, and without a replacement there is no provisioner run. The `echo` happens when the greeting changes, not when you ask for it.

Wut: Without `triggers_replace`, changing the greeting would update the resource in place and the provisioner would *not* run again. Provisioners run when a resource is created, and only then. If you want a command to run again, you have to make the resource be created again. This surprises everyone once. Try it: comment out `triggers_replace` and change the greeting twice. The first apply still echoes, because removing the trigger list is itself a replacement; the second updates in place and prints nothing.

Tip: Provisioners are OpenTofu's escape hatch, and Appendix A explains why they are a last resort in real projects. Chapter 2 replaces `echo` with resources that talk to real application programming interfaces (APIs). Even so, `terraform_data` plus `local-exec` is a fine way to glue in a command that has no provider, as Chapter 4 does with `kind load`.

Key Points

- A configuration describes an end state, not steps. Applying it twice is safe.
- The loop is `init`, `plan`, `apply`, and eventually `destroy`. Read the plan summary line first.
- State is OpenTofu's memory of what it created. Never edit it by hand, never commit it, and never lose it.
- A `resource` is a thing that should exist; `terraform_data` is a placeholder resource that exists so you can attach behavior to it.
- `variable` blocks are the inputs and `output` blocks are the return values of a configuration.
- Validation happens in layers: `fmt` for style, `validate` for structure, `validation` blocks for values, and `plan` for reality.
- Provisioners run only at creation. Use `triggers_replace` when a changed value should run them again.

New Syntax

Syntax	What it is
`resource "TYPE" "NAME" { }`	A thing that should exist, addressed as `TYPE.NAME`
`provisioner "local-exec" { command = "..." }`	A command to run when the resource is created
`# comment`	A comment, to the end of the line
`terraform { required_version = ">= 1.6" }`	Settings for OpenTofu itself
`variable "NAME" { type, default, description }`	An input, referenced as `var.NAME`
`validation { condition, error_message }`	A rule a variable's value must satisfy
`output "NAME" { value = ... }`	A value reported after apply
`"text ${expr} text"`	Interpolation: paste the value into a string
`self.ATTR`	The resource a provisioner belongs to
`triggers_replace = [ ... ]`	Replace the resource when any of these change
`length(x)`, `trimspace(s)`	Built-in functions

Try It

- Change the command to `date` and apply twice. Why does the time not update? Make it update on every apply with `triggers_replace = [timestamp()]`, and then read the plan to see what that does on every apply.
- Remove the `default` from the variable and run `tofu plan`. Then supply the value each of the three ways.

- Add a second `terraform_data` resource whose command prints the first one's `output`, and look at which one runs first. Then reverse the reference and see the order flip.
- Save a plan with `tofu plan -out=hello.tfplan` and apply it with `tofu apply hello.tfplan`. Notice that the saved plan does not ask for confirmation.
- Delete `terraform.tfstate` after an apply and run `tofu apply` again. What happens and why?

Exercises

1. **Think about it:** `tofu apply` prints hello world the first time and nothing the second time. Explain what OpenTofu compared to decide there was nothing to do, and where each side of the comparison came from.

2. **What does this do?** Given this configuration and a fresh directory, what does `tofu apply -auto-approve` print, in order?

   ```
   resource "terraform_data" "first" {
     provisioner "local-exec" {
       command = "echo one"
     }
   }

   resource "terraform_data" "second" {
     input = terraform_data.first.id

     provisioner "local-exec" {
       command = "echo two"
     }
   }
   ```

3. **Calculation:** A configuration has three `terraform_data` resources, each with `triggers_replace = [var.tag]`, and the state was applied with `tag = "a"`. What is the plan summary line for `tofu plan -var tag=b`?

4. **Where is the bug?**

   ```
   variable "name" {
     type    = string
     default = "gorgo"
   }

   resource "terraform_data" "greet" {
     provisioner "local-exec" {
       command = "echo hello var.name"
     }
   }
   ```

5. **Write a configuration** with a variable `path` that must end in `.txt` (use `endswith`), and a resource that writes the current date into that file with `date > path`. Make sure a second apply does not rewrite the file, and that changing `path` does.

Chapter 2

A Web Server on Your Machine

The need. A configuration that only runs `echo` never touches the real world. Real things live behind APIs: a container runtime, a cloud, a DNS service. OpenTofu cannot know every API, so it needs a way to load the vocabulary for each one, to pin which version of that vocabulary everyone uses, and to authenticate without putting credentials in the files. Several resources depend on one another, values have to be computed from other values, and something has to test that the result actually works.

Why it matters. Providers are how one language reaches every cloud and service, and version pinning is what makes a configuration produce the same result on a colleague's machine next year. Dependencies derived from references replace the carefully ordered steps of a script. A check that runs after every apply is the difference between "it applied" and "it works".

Why it is hard. Every provider has its own authentication and its own release cadence, and a new major version can rename arguments. The order of operations is nowhere written down; it has to be inferred from what refers to what. A change in an input, like an edited source file, has to turn into a rebuild without anyone tracking it by hand.

The strategy. Load providers through `required_providers` with constrained versions and a lock file, keep credentials in the environment, and let references between resources build the dependency graph. Compute derived values with locals and functions, and finish with a `check` block that tests the result. The example that exercises all of it is a small Go web server: the Docker provider builds its image and runs the container, and the HTTP (Hypertext Transfer Protocol) provider verifies that it answers.

The program

This example uses the simplest web server. Create the directory `webserver` for this chapter's configuration, and put this program in `webserver/app/main.go`:

```
package main

import (
    "fmt"
    "log"
    "net/http"
    "os"
)

func hello(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(w, "Hello, World!")
}
```

```
func main() {
    addr := ":8080"
    if port := os.Getenv("PORT"); port != "" {
        addr = ":" + port
    }
    http.HandleFunc("/", hello)
    log.Printf("listening on %s", addr)
    log.Fatal(http.ListenAndServe(addr, nil))
}
```

It needs a module file, `app/go.mod`, so in the `app` directory run:

```
$ go mod init motd
go: creating new go.mod: module motd
$ go mod edit -go=1.26
```

`go mod init` records the version of Go you have installed; the second command pins the file to 1.26, the version the examples and the Dockerfile use.

Check it works before involving any other tool:

```
$ go run . &
$ curl localhost:8080/
Hello, World!
```

The web server runs in a container, and `app/Dockerfile` describes its image. The image is built in two stages: one with the Go toolchain to build a static binary, and one that ships nothing but the binary:

```
# stage 1: build a static binary
FROM golang:1.26 AS build
WORKDIR /src
COPY go.mod ./
RUN go mod download
COPY . .
RUN CGO_ENABLED=0 go build -o /motd .

# stage 2: ship only the binary
FROM scratch
COPY --from=build /motd /motd
EXPOSE 8080
ENTRYPOINT ["/motd"]
```

The result is an eight megabyte image with one file in it.

Do not run `docker build` yourself. Writing the Dockerfile is all you need to do. `tofu apply` builds the image, and rebuilds it whenever the source changes.

Providers

The configuration files are in the `webserver` directory, which is the parent directory of `app`. This chapter splits the configuration across several files, which is the normal way to organize one. It starts with `versions.tf`:

```
terraform {
  required_version = ">= 1.6"

  required_providers {
```

```
    docker = {
      source  = "kreuzwerker/docker"
      version = "~> 3.0"
    }
    http = {
      source  = "hashicorp/http"
      version = "~> 3.4"
    }
  }
}
```

`required_providers` lists every provider the configuration uses. Each entry has a **source**, which is an address in the provider registry written as `namespace/name`, and a **version** constraint. The local name on the left (`docker`) is what the rest of the configuration uses, and by convention it matches the name in the source.

The `~>` operator is the **pessimistic constraint**: `~> 3.0` means "3.0 or newer, but still 3", so any 3.x release is acceptable and 4.0 is not. Providers follow semantic versioning, so a major version bump can rename arguments and break your configuration. Pinning the major version lets you take bug fixes automatically and take breaking changes on purpose.

Each provider can also have a `provider` block that configures it, in `providers.tf`:

```
provider "docker" {
  # talks to the local docker daemon; set host for a remote one, e.g.
  # host = "ssh://ubuntu@my-server"
}
```

The block is empty because the default is correct: the Docker provider talks to the local daemon socket. An empty `provider` block is optional, but it is a good place to document where the provider's settings and credentials come from.

Trap: Never put a token in a `provider` block, a variable default, or a `.tfvars` file that is committed. Providers read credentials from environment variables or from the same configuration files the vendor's own command line tools use, and that is where credentials should stay. The Docker provider needs none on your own machine; the providers in Chapter 5 read theirs from `CLOUDFLARE_API_TOKEN` and `~/.oci/config`.

Run `init` and the providers are downloaded:

```
$ tofu init
- Installing hashicorp/http v3.6.1...
- Installing kreuzwerker/docker v3.9.0...
```

Two new things appear in `webserver`. `.terraform/` holds the downloaded plugins; it is a cache and belongs in `.gitignore`. `.terraform.lock.hcl` is the **dependency lock file**: it records the exact provider versions that were selected and their checksums, so that everyone who runs `init` on this configuration gets the same providers. In a real project you commit the lock file; Appendix A has the details, including what to do for colleagues on other platforms.

Files and the graph

Every `.tf` file in `webserver` is part of the same configuration. OpenTofu takes the union of all of them, as if they were one file, and then works out the order of operations from the **references** between blocks, not from the order of files or lines. You could put every block in a single `main.tf` and nothing would change; splitting the configuration into `versions.tf`, `providers.tf`, `variables.tf`, and `main.tf` is only for humans.

Wut: There is no "top" of a configuration. You can refer to a resource defined in another file, or later in the same file, and OpenTofu builds a dependency graph from every reference. If A mentions B, then B is created before A and destroyed after it. That single rule replaces most of the ordering you would write by hand in a script.

`variables.tf` declares the one input:

```
variable "port" {
  type        = number
  description = "Host port the web server is published on."
  default     = 8080

  validation {
    condition     = var.port > 1024 && var.port < 65536
    error_message = "port must be an unprivileged port (1025-65535)."
  }
}
```

Building the image

`main.tf` begins with the image:

```
locals {
  app_dir = "${path.module}/app"
  # changes whenever any file under app/ changes
  app_hash = sha1(join("", [for f in fileset(local.app_dir, "**") :
    filesha1("${local.app_dir}/${f}")
  ]))
}

resource "docker_image" "web" {
  name = "motd:ch2"

  build {
    context = local.app_dir
  }

  triggers = {
    src = local.app_hash
  }
}
```

A `locals` block defines **local values**: named expressions. Where a variable is set by the user, a local is computed by the configuration. Use one whenever you would otherwise write the same expression twice, or when an expression needs a name. They are referenced as `local.NAME` (singular, which trips people up).

`path.module` is the directory containing the configuration, so `"${path.module}/app"` is the `app` directory no matter where you run `tofu` from.

`app_hash` is the first expression with real machinery in it, so read it from the inside out. `fileset(dir, "**")` returns the set of every file under a directory. `[for f in ... : filesha1(...)]` is a **for expression**: it produces a new list by evaluating the expression after the colon once for each element. If you know list comprehensions from Python, it is the same idea with the `for` moved to the front: Python's `[expr for x in xs]` is written `[for x in xs : expr]` here. `join("", ...)` concatenates the hashes and `sha1` boils them down to one string that changes whenever any file in `app/` changes.

`docker_image` is a resource type defined by the Docker provider, and the prefix before the first underscore is how you can tell: a type name starts with the name of the provider that defines it. That prefix is also what tells OpenTofu which plugin the block belongs to. The provider defines a family of such types, `docker_container`, `docker_network`, and `docker_volume` among them, each with its own arguments and attributes, and the provider's documentation lists all of them [1].

The `build` block tells the Docker provider to build the image from that directory instead of pulling it. `triggers` is a map of values that force a rebuild when they change, and the source hash is the natural thing to put there. Without it, editing `main.go` would not rebuild anything, because the provider has no way to know the source changed.

Running the container

Add the container to `main.tf`, after the image:

```
resource "docker_container" "web" {
  name  = "motd"
  image = docker_image.web.image_id

  ports {
    internal = 8080
    external = var.port
  }
}
```

`image = docker_image.web.image_id` is a **reference** to another resource's attribute. It does two things at once: it supplies the value, and it tells OpenTofu that the container depends on the image, so the image is built first. Attributes like `image_id` are only known after the image exists, which is why plans show them as `(known after apply)`.

`ports` is a **nested block**, a block inside a resource that groups related arguments. A resource type can allow several of the same nested block, and a container with two published ports has two `ports` blocks.

The two arguments are two sides of one door. `internal` is the port inside the container, the one the program listens on; `external` is the port on your machine that Docker connects to it. The web server always listens on 8080, so `internal` is fixed, but nothing outside the container has to know that: the container can publish it as any port you like, which is why `external` comes from a variable and `internal` does not. Set `port` to 9000 and `http://localhost:9000/` reaches a server that still thinks it is on 8080.

Checking the result

The last block in `main.tf` is a test:

```
check "hello" {
  data "http" "web" {
    url = "http://localhost:${docker_container.web.ports[0].external}/"

    retry {
      attempts     = 5
      min_delay_ms = 1000
    }
  }

  assert {
    condition     = trimspace(data.http.web.response_body) == "Hello, World!"
```

```
    error_message = "the web server did not say hello"
  }
}
```

A `check` block is a unit test for infrastructure. It runs on every plan and apply, after everything else, and reports whether an assertion holds.

Inside it is a `data` block, the first one in the booklet. A **data source** is a read-only lookup: where a resource creates something, a data source fetches something that already exists. `data "http"` fetches a URL (uniform resource locator) and exposes the `status_code` and `response_body`. Data sources are referenced with a `data.` prefix, so the body is `data.http.web.response_body`.

The URL references the container's published port, and that reference is why the check waits for the container to exist. `retry` keeps trying for a few seconds because a container does not answer the instant it starts. The `assert` block then compares the body with what you expect.

Wut: A failed `check` is a warning, not an error. Apply completes, the resources are all there, and the output says `Warning: Check block assertion failed`. That is deliberate: a check tests the result without holding the deployment hostage. When you want a failure to stop the apply, use a `precondition` or `postcondition` instead, which Chapter 3 describes.

Finally, add an output to `main.tf` that tells you where to look:

```
output "url" {
  value = "http://localhost:${var.port}/"
}
```

Running it

Initialize and plan from `webserver`:

```
$ tofu init
...
$ tofu plan
  # data.http.web will be read during apply
  # (depends on a resource or a module with changes pending)

  # docker_container.web will be created
  + resource "docker_container" "web" {
      + image = (known after apply)
      + name  = "motd"
      ...
      + ports {
          + external = 8080
          + internal = 8080
        }
    }

  # docker_image.web will be created
  + resource "docker_image" "web" {
      + image_id = (known after apply)
      + name     = "motd:ch2"
      ...
    }
```

```
Plan: 2 to add, 0 to change, 0 to destroy.

Warning: Check block assertion known after apply
```

The two resources are listed with every argument the provider knows about, most of them defaults you did not write. The container's `image` is `(known after apply)` because it refers to a resource that does not exist yet. The check's data source is deferred to apply for the same reason.

The warning is the check saying it cannot run yet because the container does not exist. Apply, and it runs at the end:

```
$ tofu apply
docker_image.web: Creating...
docker_image.web: Creation complete after 16s [id=sha256:513b4a...motd:ch2]
docker_container.web: Creating...
docker_container.web: Creation complete after 1s [id=0d901a5012b5...]

Apply complete! Resources: 2 added, 0 changed, 0 destroyed.

Outputs:

url = "http://localhost:8080/"
```

```
$ curl localhost:8080/
Hello, World!
```

Before changing any code, change the port from the command line and watch the server move:

```
$ tofu apply -auto-approve -var "port=8888"
  # docker_container.web must be replaced
-/+ resource "docker_container" "web" {
      ~ ports {
          ~ external = 8080 -> 8888 # forces replacement
            # (3 unchanged attributes hidden)
        }
    }

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

Changes to Outputs:
  ~ url = "http://localhost:8080/" -> "http://localhost:8888/"
docker_container.web: Destroying... [id=38f5e4968839...]
docker_container.web: Destruction complete after 0s
docker_container.web: Creating...
docker_container.web: Creation complete after 1s [id=4f97b176314b...]

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

Outputs:

url = "http://localhost:8888/"
$ curl localhost:8888/
Hello, World!
```

A published port cannot be changed on a running container, so the provider marks it `# forces replacement`

and the container is destroyed and created again; the image is untouched, because nothing about it changed. The `url` output follows the variable, and port 8080 no longer answers. `-auto-approve` skips the confirmation prompt, which is fine on your own laptop while you experiment and a bad habit anywhere else (Appendix A).

Changing the code

Edit `main.go` so that it says something else and run `tofu plan`:

```
$ tofu plan
  # docker_container.web must be replaced
-/+ resource "docker_container" "web" {
      ~ image = "sha256:513b4a..." -> (known after apply) # forces replacement
      ...
    }

  # docker_image.web must be replaced
-/+ resource "docker_image" "web" {
      ~ triggers = { # forces replacement
          ~ "src" = "97bf81ab..." -> "b860b2aa..."
        }
    }

Plan: 2 to add, 0 to change, 2 to destroy.
```

The source hash changed, so the image is replaced; the image id will change, so the container is replaced. Run `tofu apply`, and the old server shuts down while a new one starts. That plan is the whole point of the chapter: from one edit to a Go file, OpenTofu worked out the two things that have to happen.

Tip: If the container fails to start with `Bind for 0.0.0.0:8080 failed: port is already allocated`, something else on your machine owns the port. Apply again with `-var port=8081`. The variable is there so that you do not have to edit the configuration to work around your laptop.

Shutting it down

When you are done, `destroy` takes down everything the configuration created, in the reverse of the order it was created:

```
$ tofu destroy
  # docker_container.web will be destroyed
  # docker_image.web will be destroyed

Plan: 0 to add, 0 to change, 2 to destroy.

Do you really want to destroy all resources?
  Enter a value: yes

docker_container.web: Destroying... [id=61b2a04defb3...]
docker_container.web: Destruction complete after 1s
docker_image.web: Destroying... [id=sha256:513b4a...motd:ch2]
docker_image.web: Destruction complete after 0s
```

```
Destroy complete! Resources: 2 destroyed.
```

The container goes first because it depends on the image, then the image. Afterwards `docker ps -a` and `docker image ls` show no trace of `motd`, and the state file records no resources. The source in `app` is untouched; only what the configuration created is gone.

Trap: `tofu destroy` destroys everything in the state, and in later chapters that includes a database full of data and a machine in the cloud. For things you want to keep, add `lifecycle { prevent_destroy = true }` to the resource, which makes any plan that would destroy it fail. Alternatively, `tofu state rm ADDRESS` makes OpenTofu forget a resource without deleting it, and a later `tofu apply` will then try to create it again.

Key Points

- A provider is a plugin that adds resource types by translating them into API calls. `required_providers` says which ones you need; `init` downloads them; the lock file pins them.
- Credentials come from the environment or the vendor's own config files, never from your files.
- All `.tf` files in a directory are one configuration, and references between blocks determine the order of operations.
- Locals name computed values; variables hold user-supplied ones.
- A reference to another resource's attribute both supplies a value and creates a dependency.
- Data sources read things that exist; resources create things.
- `check` blocks test the result of an apply and warn when it is wrong.
- Content hashes turn "the source changed" into "the image must be rebuilt" without anyone tracking it by hand.

New Syntax

Syntax	What it is
`required_providers { NAME = { source, version } }`	Which plugins the configuration needs
`version = "~> 3.0"`	Pessimistic constraint: any 3.x
`provider "NAME" { }`	Configuration for a provider
`locals { NAME = expr }`	Named expressions, referenced as `local.NAME`
`path.module`	Directory containing the configuration
`[for x in list : expr]`	For expression: build a list from a list
`fileset`, `filesha1`, `join`, `sha1`	File and hash functions
`TYPE.NAME.ATTR`	Reference to a resource attribute (and a dependency)
`ports { ... }`	A nested block inside a resource
`data "TYPE" "NAME" { }`	A read-only lookup, referenced as `data.TYPE.NAME`
`check "NAME" { data ...; assert { condition, error_message } }`	A test that runs after every plan and apply
`list[0]`	Indexing into a list

Try It

- Change the container to restart automatically with `restart = "unless-stopped"`, reboot Docker, and confirm the server comes back.

- Add `lifecycle { prevent_destroy = true }` to the container and run `tofu plan -destroy`.
- Make the check fail on purpose by changing the expected string, and watch apply complete anyway. Then look at `tofu show` and find the check result in the state.
- Put the `check` block in its own file, `checks.tf`, and confirm nothing changes. Files are just organization.
- Put `webserver` in a git repository and add a GitHub Actions workflow that runs `tofu fmt -check` and `tofu validate` on every push. Both need no credentials.

Exercises

1. **Think about it:** The image resource has a `triggers` map holding a hash of the source directory. What would happen on the second `apply` if the hash were replaced with `timestamp()`? What would happen if `triggers` were removed entirely and you edited `main.go`?

2. **What does this do?** For the expression below, what is the type of the result and how many elements does it have when `app/` contains `main.go`, `go.mod`, and `Dockerfile`?

   ```
   [for f in fileset("app", "*.go") : upper(f)]
   ```

3. **Calculation:** A configuration pins `version = "~> 3.4"` for a provider whose available releases are 3.3.0, 3.4.2, 3.9.0, and 4.0.1. Which release does `tofu init` install, and which is the newest it could ever install without editing the constraint?

4. **Where is the bug?**

   ```
   resource "docker_container" "web" {
     name  = "motd"
     image = docker_image.web.name

     ports {
       internal = 8080
       external = var.port
     }
   }
   ```

5. **Where is the bug?**

   ```
   locals {
     app_dir = "${path.module}/app"
   }

   resource "docker_image" "web" {
     name = "motd:ch2"
     build {
       context = locals.app_dir
     }
   }
   ```

6. **Write a configuration** that runs two copies of the web server on ports 8080 and 8081 from the same image, with a check for each. Then reduce the duplication with a variable of type `list(number)` and `count` (look it up), and compare the plans.

Chapter 3

A Database of Sayings

The need. A configuration does not only wire values through; sometimes it has to produce them: generate data, render a file, make a password. Secrets must exist without appearing in any file you edit, and must be marked so that OpenTofu keeps them off the screen. Some dependencies are invisible to providers, like "new seed data means a new volume", and some assumptions should fail the plan rather than fail an hour later. And "created" must mean "ready", or everything downstream starts too early.

Why it matters. Expressions and templates keep the configuration the single source of truth for derived data, instead of a checked-in file that drifts. The `random` provider and `sensitive` are how a secret is generated once and then handled carefully wherever it goes. Lifecycle rules, preconditions, and explicit dependencies are the tools for the ordering and rebuild logic that providers cannot infer.

Why it is hard. Generated data has to be escaped, counted, and checked before it can be trusted. A secret in state is still a secret, and OpenTofu's notion of sensitivity ends at the terminal. Replacement chains have to be stated exactly, or a change rebuilds too little (stale data) or too much (a rebuilt web server for every edit). Readiness is not existence, and a provider has to be told what "healthy" means.

The strategy. Use a module from this booklet's public repository that combines a list of prefix and suffix phrases with `setproduct` and a `for` to generate a list used to render the SQL with `templatefile`. Write the SQL from the module using `local_file`, and guard it with a `precondition`. Generate the password with `random_password`, mark it sensitive, and pass it only where it is needed. State the invisible dependency with `replace_triggered_by`, the ordering with `depends_on`, and readiness with a health check that creation waits on. The example is a MySQL database seeded with 3600 sayings, one per second of the hour, and the web server from Chapter 2 rewritten to look up the saying for the current time.

Two more providers

Chapter 2 used one provider that talks to a real system. This chapter adds two that do not talk to anything: they exist so that a configuration can produce values and files of its own. Neither is built into OpenTofu; both come from the registry, so `versions.tf` in a new `database` directory needs two more entries:

```
terraform {
  required_version = ">= 1.6"

  required_providers {
    docker = {
      source  = "kreuzwerker/docker"
      version = "~> 3.0"
    }
    random = {
```

```
      source  = "hashicorp/random"
      version = "~> 3.6"
    }
    local = {
      source  = "hashicorp/local"
      version = "~> 2.5"
    }
    http = {
      source  = "hashicorp/http"
      version = "~> 3.4"
    }
  }
}

provider "docker" {}
```

The **random** provider generates values: `random_password` and `random_string` for text, `random_integer`, `random_id`, `random_uuid`, and `random_pet` for the kind of names Docker gives containers [2]. What makes them resources rather than functions is memory. A function like `uuid()` gives a new answer every time it is evaluated; a `random_password` resource generates its value once, when it is created, stores it in state, and hands back the same value on every later plan until the resource is replaced. That is the property a password needs, and `random_password` marks its `result` as sensitive so that it stays off the screen.

The **local** provider works with files on the machine running `tofu`: `local_file` writes a file with a given `content` to a given `filename`, and `local_sensitive_file` does the same without echoing the content in plans [3], [4]. The file is a resource like any other: created on apply, replaced when its content changes, and deleted on destroy. This chapter uses it to write the SQL that seeds the database, so that the seed is generated from the configuration rather than checked in by hand.

The lifecycle block

Every argument you have written so far belonged to a resource type and was handed to its provider. A few arguments belong to OpenTofu instead, and are accepted by every resource regardless of type; these are the **meta-arguments**, and `depends_on` from Chapter 2 is one of them. The `lifecycle` block is another [5] with these settings:

Setting	What it tells OpenTofu
`create_before_destroy = true`	When replacing, build the new one before removing the old
`prevent_destroy = true`	Refuse any plan that would destroy this
`ignore_changes = [ ... ]`	Do not plan updates for these arguments when they drift
`replace_triggered_by = [ ... ]`	Replace this whenever those resources change
`precondition { }`	An assumption checked before the resource is planned
`postcondition { }`	A guarantee checked after the resource is applied

The way to think about them: a provider knows how its own resource type behaves, but it cannot know how your resources relate to each other or what you are assuming about them. `lifecycle` is where you write that down. `replace_triggered_by` expresses a relationship the provider cannot see, like new seed data meaning

a new volume. `precondition` and `postcondition` express assumptions and guarantees as expressions that fail the plan or the apply with your own error message [6]. `prevent_destroy` and `ignore_changes` express intent about how the resource may change. This chapter uses `replace_triggered_by` and `precondition`; the others appear in Appendix A's advice on stateful resources and drift.

Pick your own configuration file names

Hopefully, you are comfortable with the idea that configurations can be spread across configuration files in the same directory. Use the `database` directory for this chapter. Rather than telling you the names of `.tf` files to use, you can pick your own. Organize it how you see fit. You can put them all in one big file, or you can have a `.tf` file for each aspect of the project. From now on, it is up to you. You'll see the configuration to add; you can put it in the file of your choice.

A module from GitHub

The sayings are built from two word lists of sixty lines each, which live in this booklet's repository on GitHub, in `examples/motd`. Rather than copy them, the configuration pulls them from there with a `module` block:

```
module "sayings" {
  source = "github.com/BooksByGorgo/opentofu//tasting/examples/motd?ref=main"
}
```

A **module** is a directory of configuration with inputs and outputs. The directory you run `tofu` in is the **root module**; any directory it pulls in with a `module` block is a **child module**. A module is a function: its variables are the parameters, its resources are the body, and its outputs are the return values. Calling one means writing a `module` block, and the call's result is the child's outputs, available as `module.NAME.OUTPUT`.

The `source` says where the directory comes from [7]: a GitHub repository, `//` followed by the path inside it, and `?ref=` naming the branch, tag, or commit to take. The `//` marks where the repository address ends and the path inside it begins: OpenTofu clones everything to the left of it, then descends to the directory on the right, which is why `?ref=` sits at the end of the line but chooses the branch to clone. Without that separator nothing would say whether `opentofu/tasting` is a longer repository name or a directory inside a shorter one.

Wut: For a `github.com/` source the `//` is optional. OpenTofu knows that such an address is `github.com/OWNER/REPO`, and treats whatever follows as the path inside the repository, so a single slash initializes exactly the same way. Write the `//` anyway: an explicit `git::https://...` source gets no such help, and without the separator the whole path is taken as the repository and the clone fails with `Failed to download module`.

Local paths and the module registry are the other kinds of source, and Chapter 4 uses the first of them. `tofu init` clones the repository over HTTPS (HTTP Secure), which for a public repository needs no token and no account, and puts the directory under `.terraform/modules`:

```
$ tofu init
Initializing modules...
Downloading git::https://github.com/BooksByGorgo/opentofu.git?ref=main for sayings...
- sayings in .terraform/modules/sayings/tasting/examples/motd
...
```

Here is what it fetched, `examples/motd/main.tf`. **You do not need to add it to a config file since OpenTofu will pull it in.**

```
# A data-only module: no variables, no resources, four outputs.
# Any configuration can pull it straight from this public repository with
```

```
#   source = "github.com/BooksByGorgo/opentofu//tasting/examples/motd?ref=main"
# 60 prefixes x 60 suffixes = 3600 sayings, one for every second of the hour.
# Saying number n pairs prefix n / 60 with suffix n % 60.

locals {
  prefixes = split("\n", trimspace(file("${path.module}/prefix.txt")))
  suffixes = split("\n", trimspace(file("${path.module}/suffix.txt")))

  # setproduct pairs every prefix with every suffix, prefixes first:
  # [[p0, s0], [p0, s1], ..., [p1, s0], ...]
  sayings = [
    for pair in setproduct(local.prefixes, local.suffixes) :
    "${pair[0]} ${pair[1]}"
  ]
}

output "prefixes" {
  description = "The 60 sentence beginnings, one per line in prefix.txt."
  value       = local.prefixes
}

output "suffixes" {
  description = "The 60 sentence endings, one per line in suffix.txt."
  value       = local.suffixes
}

output "sayings" {
  description = "The 3600 sayings, indexed by second of the hour."
  value       = local.sayings
}

output "sql" {
  description = "SQL that creates and fills the sayings table."
  value = templatefile("${path.module}/sayings.sql.tftpl", {
    sayings = local.sayings
  })

  precondition {
    condition     = length(local.sayings) == 3600
    error_message = "need exactly 3600 sayings, one per second of the hour."
  }
}
```

This module has no variables and no resources at all; it is a function with no parameters that returns data. The two `locals` read the word lists with `file` and split them into one entry per line; `path.module` inside a module is the module's own directory, not the root's, even when that directory is a checkout under `.terraform/modules`. The rest of the file is outputs, and outputs are all a caller can see: the two lists, the 3600 sayings that `setproduct` makes of them, and the finished SQL rendered by `templatefile`. Both functions are explained below, `setproduct` next and `templatefile` in the seed file section. The `precondition` sits on the `sql` output, which is where that assumption belongs.

Tip: `?ref=main` takes whatever is on the branch today, which is fine for a booklet and wrong for anything that must build the same way twice. A module source is code you run, so in real work pin `ref` to a tag or a commit hash, and move it on purpose.

The sayings

Writing 3600 sayings by hand is not going to happen. Sixty sentence beginnings (`A patient programmer`, `The wise sysadmin`, ...) and sixty endings (`never blames the compiler.`, `starts every day with a backup.`, ...) are 120 lines of input. Pairing every beginning with every ending turns them into 3600 sayings, and the results have the earnest nonsense quality of a real fortune file.

`setproduct` returns every combination of its lists as a list of pairs, first list varying slowest. The for expression turns each pair into one string. Saying number `n` is prefix `n / 60` with suffix `n % 60`, so with `n = minute * 60 + second` the beginning changes every minute and the ending every second.

This is all done in the `sayings` module. That module outputs the sayings as `sayings`. They can be accessed as a resource using `module.sayings.sayings`.

When an expression gets this clever, check it. `tofu console` is an interactive prompt that evaluates expressions against the configuration, and once `init` has fetched the module it can evaluate these:

```
$ tofu console
> length(module.sayings.sayings)
3600
> module.sayings.sayings[0]
"A patient programmer never blames the compiler."
> module.sayings.sayings[61]
"The wise sysadmin starts every day with a backup."
> module.sayings.sayings[3599]
"A brand-new laptop makes tomorrow easier than today."
```

Tip: `tofu console` is the fastest way to learn what a function does or to debug an expression. Type it, see the result, adjust. It can also read resource attributes from the state once you have applied, so `docker_container.web.ports` will show you exactly what the provider recorded.

The seed file

The database needs SQL to populate the table, and the SQL is generated from the list with a **template**, `sayings.sql.tftpl`. **Again, the module does this for you, so this is just a peek into what it is doing.**

```
CREATE TABLE sayings (
  id     INT PRIMARY KEY,
  saying VARCHAR(255) NOT NULL
);

INSERT INTO sayings (id, saying) VALUES
%{ for i, s in sayings ~}
  (${i}, '${replace(s, "'", "''")}')${i < length(sayings) - 1 ? "," : ";"}
%{ endfor ~}
```

A template is a fill-in-the-blanks file processed by the `templatefile` function. `.tftpl` is the conventional extension for one. That extension can be anything except `.tf`, since OpenTofu parses every `.tf` file in the directory as configuration. The function fills in the blanks two ways. `${...}` you already know: paste a

value. `%{ for ... }` and `%{ endfor }` are **directives**: they repeat or choose the text between them. The `~` on a directive eats the newline after it, so that the loop does not emit a blank line for every iteration. `for i, s in sayings` gives you the zero-based index as well as the element.

Two more expressions are hiding in that line. `replace(s, "'", "''")` doubles any single quote, which is how SQL escapes them; none of the sayings has one, but the template should not break when you add one. `cond ? a : b` is the **conditional expression**: a comma after every row except the last, which gets the semicolon.

The module's `sql` output is that SQL, already rendered. You need to copy that SQL somewhere that OpenTofu can find it to populate in the database image. Add this `local_file` resource block to write the SQL to a file that OpenTofu can use to build the database docker image:

```
resource "local_file" "seed" {
  filename = "${path.module}/seed/001-sayings.sql"
  content  = module.sayings.sql

  lifecycle {
    precondition {
      condition     = length(module.sayings.sayings) == 3600
      error_message = "need exactly 3600 sayings, one per second of the hour."
    }
  }
}
```

The `precondition` checks assumptions: this configuration makes no sense unless there is one saying for every second of the hour. A wrong module version, or a word list that lost a line, fails the plan with your message rather than filling the database with holes.

A precondition is evaluated on the machine running `tofu`, from the configuration and the state, at the point OpenTofu plans the resource. The precondition does not have access to Docker or MySQL. It is evaluated on every plan, not only the one that creates the file, so the assumption keeps holding after the seed exists.

Trap: A precondition only fails the plan when it can be evaluated during the plan. If the condition reads a value that is not known until apply, such as a generated password or an id the provider assigns, OpenTofu defers the check to the apply, where it fails after earlier resources have already been created. `length(module.sayings.sayings)` is known during the plan, because the module only reads files, so this one fails before anything is built.

The password

This resource block will create a random password for your database.

```
resource "random_password" "db" {
  length  = 24
  special = false # keeps the password safe to paste into a DSN
}
```

As you saw earlier, the value is generated once and remembered in state, so the password is the same on every plan until you replace the resource. That is what you want: a password that is generated once, never typed, and never written into a file you edit.

Wut: The "random" provider is only random once. The generated value is stored in the state file, and every later run reads it back from there. That makes the state file a secret, and it makes `tofu state rm random_password.db` the way to force a new password (the next apply generates one and updates everything that references it).

The password shows up in the outputs marked as sensitive; it will not display in the plan:

```
output "db_password" {
  value     = random_password.db.result
  sensitive = true
}
```

A sensitive output is printed as `(sensitive value)` after apply, but `tofu output -raw db_password` still gives you the real thing when you ask for it explicitly. Any value derived from a sensitive value is sensitive too, so the container's environment list is hidden in plans as well.

Trap: `sensitive` hides values from the terminal, not from the state file. The password is in `terraform.tfstate` in plain text. This is one more reason state never goes into git and, once anyone else needs it, lives in a backend with access control (Appendix A).

Network and volume

These resources set up docker network and storage:

```
resource "docker_network" "motd" {
  name = "motd"
}

resource "docker_volume" "db_data" {
  name = "motd-db-data"

  lifecycle {
    # the seed only loads into an empty volume, so new seed => new volume
    replace_triggered_by = [local_file.seed]
  }
}
```

Containers on the same user-defined Docker network can reach each other by name, which is how the web server will find the database. The next section shows where those names come from: each container's `name` argument, plus any `aliases` it asks for on the network. The name resolves to the container's current address, handed out when it starts and different after every replacement, so the name is the stable half and no file in this chapter mentions an IP address. The volume holds the database files so that replacing the database container does not lose the data.

`replace_triggered_by` is the second lifecycle setting in this chapter and it solves a real problem. MySQL runs the seed scripts only when its data directory is empty. If the sayings or the template change, the seed file changes, but a volume full of old data would ignore it. This rule says: whenever `local_file.seed` is replaced, replace the volume too, and a fresh volume gets the fresh seed.

The database container

These resources set up the docker image and container for the database:

```
resource "docker_image" "mysql" {
  name = "mysql:8.4"
}

resource "docker_container" "db" {
  name  = "motd-db"
  image = docker_image.mysql.image_id

  env = [
    "MYSQL_RANDOM_ROOT_PASSWORD=yes",
    "MYSQL_DATABASE=motd",
    "MYSQL_USER=motd",
    "MYSQL_PASSWORD=${random_password.db.result}",
  ]

  networks_advanced {
    name    = docker_network.motd.name
    aliases = ["db"]
  }

  volumes {
    volume_name    = docker_volume.db_data.name
    container_path = "/var/lib/mysql"
  }

  volumes {
    host_path      = abspath(dirname(local_file.seed.filename))
    container_path = "/docker-entrypoint-initdb.d"
    read_only      = true
  }

  healthcheck {
    test     = ["CMD", "mysqladmin", "ping", "-h", "127.0.0.1", "--silent"]
    interval = "5s"
    timeout  = "3s"
    retries  = 3
  }
  wait         = true # creation finishes when the health check passes
  wait_timeout = 180

  lifecycle {
    replace_triggered_by = [docker_volume.db_data]
  }
}
```

The `docker_image` without a `build` block pulls the image from Docker Hub. The environment variables are the MySQL image's own interface: it creates the database and the user on first start. `MYSQL_RANDOM_ROOT_PASSWORD` gives the root account a password nobody knows, printed once to the container log and used by nothing in this configuration, since the web server connects as `motd`.

The `networks_advanced` block joins the network and gives the container the alias `db`, so that the web server can use `db` as a host name without caring what the container is called.

Two `volumes` blocks mount two things: the named volume for the data, and the directory holding the seed file.

That second mount is how the sayings reach the database, and nothing in this configuration executes them. `/docker-entrypoint-initdb.d` is a convention of the MySQL image: on its first start, before it accepts connections from the network, its entrypoint script runs every `.sql` and `.sh` file it finds in that directory, in filename order, and then never looks again. The number in `001-sayings.sql` is there so that a second seed file sorts after the first. "First start" means an empty data directory, which is the volume from the previous section, and that is why a new seed needs a new volume.

Docker needs an absolute path for a bind mount, so `dirname` takes the seed file's directory and `abspath` makes it absolute. Referencing `local_file.seed.filename` also makes sure the file is written before the container starts.

The `healthcheck` block defines what "healthy" means, and `wait = true` makes the provider hold the resource in "Creating" until the health check passes. Pinging `127.0.0.1` rather than `localhost` matters, because a MySQL client treats `localhost` as an instruction to use the Unix socket rather than as a host name. While MySQL is loading seed scripts it runs a temporary server that listens on that socket only, so a `localhost` ping would report healthy with the rows still loading, while a ping over TCP (Transmission Control Protocol) fails until the real server is up. So when this resource is "created", the sayings are loaded and the database is accepting connections.

The final `replace_triggered_by` completes the chain: seed file replaced, so volume replaced, so container replaced.

The web server

The Go program needs a database lookup. Copy `app` from `webserver` into `database`; change `app/main.go` to match this:

```
package main

import (
    "database/sql"
    "fmt"
    "log"
    "net/http"
    "os"
    "time"
    _ "time/tzdata" // zone database, since a scratch image has none

    _ "github.com/go-sql-driver/mysql"
)

var db *sql.DB

// saying looks up the message for the current second of the hour.
func saying(w http.ResponseWriter, r *http.Request) {
    now := time.Now()
    key := now.Minute()*60 + now.Second()
    var text string
    err := db.QueryRow("SELECT saying FROM sayings WHERE id = ?", key).Scan(&text)
    if err != nil {
        log.Printf("lookup %d: %v", key, err)
        http.Error(w, "no saying right now, try again", http.StatusServiceUnavailable)
        return
    }
    fmt.Fprintf(w, "[%s] %s\n", now.Format("15:04:05"), text)
```

```
}

func main() {
    addr := ":8080"
    if port := os.Getenv("PORT"); port != "" {
        addr = ":" + port
    }
    var err error
    db, err = sql.Open("mysql", os.Getenv("DB_DSN")) // connects lazily
    if err != nil {
        log.Fatal(err)
    }
    db.SetConnMaxLifetime(3 * time.Minute)
    http.HandleFunc("/", saying)
    log.Printf("listening on %s", addr)
    log.Fatal(http.ListenAndServe(addr, nil))
}
```

Two things to note about the container. `sql.Open` does not connect until the first query, so the server starts even if the database is not ready, and answers 503 until it is. And `time/tzdata` embeds the time zone database, because the `scratch` image has no `/usr/share/zoneinfo`, and without it `TZ=America/Los_Angeles` silently means Coordinated Universal Time (UTC).

Add the driver with `go get github.com/go-sql-driver/mysql`, which updates `go.mod` and creates `go.sum`. Then change the Dockerfile's `COPY go.mod ./` to `COPY go.mod go.sum ./`.

The container for the docker image of the app reuses the image build from Chapter 2, so move that over, and add an environment and a network:

```
resource "docker_container" "web" {
  name  = "motd-web"
  image = docker_image.web.image_id

  env = [
    "DB_DSN=motd:${random_password.db.result}@tcp(db:3306)/motd",
    "TZ=${var.timezone}",
  ]

  networks_advanced {
    name = docker_network.motd.name
  }

  ports {
    internal = 8080
    external = var.port
  }

  depends_on = [docker_container.db]
}
```

The DSN (data source name) names the host `db`, which is the alias on the shared network, and pastes in the generated password. `timezone` is a new variable. You will need that as well as the port.

```
variable "port" {
  type        = number
  description = "Host port the web server is published on."
  default     = 8080
```

```
}

variable "timezone" {
  type        = string
  description = "Time zone the server uses to pick the current saying."
  default     = "America/Los_Angeles"
}
```

Set `timezone` to your own zone. (`America/Los_Angeles` for the west coast.)

`depends_on` is an **explicit dependency**. Nothing in this block references the database container, so OpenTofu would happily start the web server first. `depends_on` says: wait for that. Use it only when there is no attribute you can reference, because a reference is both a dependency and documentation of why.

An apply can finish with every resource created and the application still broken: the containers run and MySQL reports healthy, but the page answers 503 if the seed never loaded or the password in the DSN is wrong. A `check` block is what allows you to do more than just tell if it applied; you can test that it works. Once everything has been configured, the check allows you to verify that things are working.

```
check "saying" {
  data "http" "web" {
    url = "http://localhost:${docker_container.web.ports[0].external}/"

    retry {
      attempts     = 5
      min_delay_ms = 2000
    }
  }

  assert {
    condition = can(regex(
      "^\\[\\d\\d:\\d\\d:\\d\\d\\] .+", data.http.web.response_body
    ))
    error_message = "not a saying: ${data.http.web.response_body}"
  }
}
```

The check runs on the machine running `tofu`, not inside Docker: a provider is a plugin that OpenTofu runs as a local process, which is why the URL is `localhost` and the container's published port rather than the `db`-style name the web server uses on the network. The `assert` condition is evaluated in the same place, against the response already fetched.

Three guards in this chapter answer three different questions, and they run in three different places:

Guard	Where it runs	When
`precondition`	the machine running `tofu`	plan, or apply if a value is unknown
`healthcheck`	inside the container, by Docker	while the container starts
`check`	the machine running `tofu`	after apply

A precondition stops the work before it starts, a health check decides when "created" means "ready", and a check reports on the result without failing the apply.

`regex` returns the match or raises an error when there is none, and `can` turns "raises an error" into `false`. `can(regex(...))` is the idiom for "does this match". Note the doubled backslashes: the string is parsed once by the configuration language and once by the regular expression engine.

Running it

From the `database` directory:

```
$ tofu init
...
$ tofu apply
random_password.db: Creation complete after 0s [id=none]
docker_network.motd: Creation complete after 3s [id=58234eb2ca78...]
local_file.seed: Creation complete after 9s [id=ed5371baf622...]
docker_volume.db_data: Creation complete after 0s [id=motd-db-data]
docker_image.web: Creation complete after 17s [id=sha256:9c956a29...motd:ch3]
docker_image.mysql: Creation complete after 41s [id=sha256:bced325a...mysql:8.4]
docker_container.db: Creating...
docker_container.db: Still creating... [10s elapsed]
docker_container.db: Creation complete after 18s [id=c6fa127e59d8...]
docker_container.web: Creation complete after 1s [id=794f91260fc7...]

Apply complete! Resources: 8 added, 0 changed, 0 destroyed.
```

Read the order. Everything with no dependencies started at once; the database waited for the network, the volume, and the seed; it then spent eighteen seconds loading 3600 rows before it counted as created; and only then did the web server start. None of that order was written down anywhere.

```
$ curl localhost:8080/; sleep 1; curl localhost:8080/
[09:27:12] A well-named variable knows that naming things is harder.
[09:27:13] A well-named variable counts from zero.
```

Minute 27 is prefix 27, and the suffix walks along with the seconds.

Now change the seed: put a comment line such as `-- version 2` at the top of the template, and plan:

```
$ tofu plan
  # docker_container.db will be replaced due to changes in replace_triggered_by
  # docker_volume.db_data will be replaced due to changes in replace_triggered_by
  # local_file.seed must be replaced

Plan: 3 to add, 0 to change, 3 to destroy.
```

The seed, the volume, and the database are rebuilt; the web server, its image, the network, and the password are untouched.

Trap: A container that was started by hand with `docker run --name motd-db` makes apply fail with `Conflict. The container name "/motd-db" is already in use`. OpenTofu only knows about containers it created. Remove the stray one, or `tofu import` it (Appendix A), and apply again.

Key Points

- A module is a directory of configuration called with a `module` block; its outputs are what the caller sees, and a module with no resources is still a useful function.
- A module `source` can be a public GitHub repository, with `//` for the path inside it and `?ref=` to pin what you get.

- `setproduct` and for expressions generate data; `templatefile` with `%{ for }` directives renders it; `local_file` writes it.
- `precondition` blocks turn assumptions into errors before a resource is created; `check` blocks turn results into warnings after.
- The `random` provider generates a value once and keeps it in state. Mark such values `sensitive`, and remember that state holds them in plain text.
- Containers on a user-defined network find each other by name or alias. Named volumes keep data across container replacement.
- The seed loads because `local_file` writes it into the directory bind-mounted at `/docker-entrypoint-initdb.d`. The MySQL image runs the files there itself, on first start only and in filename order; OpenTofu writes the file and nothing else.
- `replace_triggered_by` chains replacements that the provider cannot see, like "new seed, so new volume".
- A `healthcheck` plus `wait = true` makes "created" mean "ready", which is what everything downstream needs.
- `depends_on` orders resources that do not reference each other.
- Errors in expressions can be caught with `can()`.

New Syntax

Syntax	What it is
`module "NAME" { source = "github.com/OWNER/REPO//DIR?ref=REF" }`	Pull a module from a GitHub repository
`module.NAME.OUTPUT`	Read a module output
`setproduct(a, b)`	Every pair from two lists
`templatefile(path, { vars })`	Render a template with the given values
`%{ for i, x in list ~} ... %{ endfor ~}`	Template loop directive; ~ eats the newline
`cond ? a : b`	Conditional expression
`replace`, `abspath`, `dirname`, `regex`, `can`	More built-in functions
`lifecycle { precondition { } }`	An assumption that fails the plan when false
`lifecycle { replace_triggered_by = [ ] }`	Replace this when those are replaced
`sensitive = true`	Hide an output or variable from the terminal
`depends_on = [ ]`	Explicit ordering without a reference
`healthcheck { }` with `wait = true`	Creation completes when the container is healthy
`tofu console`	Evaluate expressions interactively

Try It

- Change `?ref=main` to the hash of a commit in the booklet's repository, run `tofu init`, and see what `.terraform/modules/modules.json` records.
- Replace the generated sayings with real ones from a fortune file: on many Linux systems there is one at `/usr/share/games/fortunes/fortunes`, with entries separated by `%`. You need exactly 3600, so decide what to do when there are too few or too many.
- Add a `/healthz` endpoint that runs `db.Ping()`, and point the check at it.
- Change `timezone` to `UTC` and apply. Which resources change, and why does the database not?
- Use the `petoju/mysql` provider to create a second, read-only user for the web server, so that the user created by the image's environment variables is only used for setup.
- Change `special = false` to `special = true` on the password and find out what breaks.
- Add a second seed file, `002-extra.sql`, with one more `INSERT`, and apply. The table now holds 3601 rows and the precondition still passes. Work out why, and what it would have to count instead.

Exercises

1. **Think about it:** The database container has `wait = true` and the web container has `depends_on = [docker_container.db]`. What would go wrong if you removed only `wait`? What if you removed only `depends_on`?

2. **What does this do?** What does the template below produce for `names = ["a", "b", "c"]`?

   ```
   %{ for i, n in names ~}
   ${i}:${n}${i < length(names) - 1 ? "," : ""}
   %{ endfor ~}
   ```

3. **Calculation:** At 14:05:09 local time, which saying does the server return, as a prefix index and a suffix index? Which second of which minute returns `module.sayings.sayings[3599]`?

4. **Where is the bug?**

   ```
   resource "docker_container" "db" {
     name  = "motd-db"
     image = docker_image.mysql.image_id

     volumes {
       host_path      = "./seed"
       container_path = "/docker-entrypoint-initdb.d"
     }
   }
   ```

5. **Where is the bug?**

   ```
   output "dsn" {
     value = "motd:${random_password.db.result}@tcp(db:3306)/motd"
   }
   ```

6. **Write a configuration** that adds a second table, `visits`, seeded from a template with one row per weekday, and a second `check` that queries the database through `docker exec` in a `terraform_data` provisioner. Decide whether the provisioner or an `http` data source is the better tool, and say why.

Chapter 4

Kubernetes

The need. A configuration that has grown past a screenful needs the same thing a program does at that size: functions. Groups of resources that belong together should be defined once, take parameters, return values, and be called from more than one place. Some parameters are optional and turn whole blocks on or off. And as the number of objects grows, OpenTofu needs a reliable signal that something changed, so that an updated artifact becomes a visible plan line and not a silent no-op.

Why it matters. Modules are how a configuration is reused across environments instead of copied, and how a team divides responsibility: the root decides where things go, a module decides what goes there. Optional blocks are what let one module serve callers with different needs. A content-addressed name turns "rebuild" into a change OpenTofu can plan, which is the difference between a rollout and a stale deployment.

Why it is hard. A module has to be general enough to reuse and specific enough to stay simple, and nested blocks cannot be made conditional without new syntax. Providers are configured in one place and used in another, which is a rule that has to be learned. Objects depend on each other by name, images have to reach the cluster, storage binds lazily, and readiness has to be defined before OpenTofu can wait for it.

The strategy. Write an application module with variables, outputs, and `dynamic` blocks for its optional parts, called from a root module that supplies the providers, the way Chapter 3 called the sayings module. Name the image by the hash of its source so that a rebuild changes the deployment. Let readiness probes sequence the rollout, the same idea as the health check in Chapter 3. The example is the sayings server deployed to a local Kubernetes cluster from `kind`, as a namespace, a secret, a config map, a volume claim, two deployments, and two services.

The Kubernetes provider

The configuration in `kubernetes` swaps Chapter 3's `local` provider for the Kubernetes provider, since the seed goes into a config map now rather than a file; `versions.tf`:

```
terraform {
  required_version = ">= 1.6"

  required_providers {
    docker = {
      source  = "kreuzwerker/docker"
      version = "~> 3.0"
    }
    kubernetes = {
      source  = "hashicorp/kubernetes"
      version = "~> 2.35"
```

```
    }
    random = {
      source  = "hashicorp/random"
      version = "~> 3.6"
    }
    http = {
      source  = "hashicorp/http"
      version = "~> 3.4"
    }
  }
}

provider "docker" {}

provider "kubernetes" {
  config_path    = "~/.kube/config"
  config_context = "kind-motd"
}
```

The Kubernetes provider has a resource type for every kind of Kubernetes object, named after it [8]. This chapter uses six: `kubernetes_namespace_v1`, `kubernetes_secret_v1`, `kubernetes_config_map_v1`, `kubernetes_persistent_volume_claim_v1`, `kubernetes_deployment_v1`, and `kubernetes_service_v1`. The `_v1` suffix picks the newer implementation of each, which follows the Kubernetes API more strictly; use those for anything new. Every one of them has a `metadata` block for the name, namespace, and labels, and most have a `spec` block whose contents mirror the YAML you would otherwise write, so if you know the YAML you know the arguments, and the provider's documentation gives the argument for each Kubernetes field.

Three things about how the provider behaves matter here. It waits: a deployment counts as created when its rollout completes and a volume claim when it is bound, and a `timeouts` block on the resource says how long to wait, since `timeouts` is defined by the provider rather than by OpenTofu [9]. Its `provider` block points at a cluster through a kubeconfig file and a context name, so the same configuration can target another cluster by changing two lines. And it cannot put images into a cluster; the cluster pulls them, which is why getting the image there needs a step of its own.

A cluster to deploy to

`kind` runs a Kubernetes cluster inside Docker containers. This chapter's configuration goes in `kubernetes`, with the modules in a sibling directory, `modules`. Install `kind` and `kubectl` (Chapter 0 has the commands for each system), then create a cluster from `kubernetes/kind-config.yaml`:

```yaml
# a one-node cluster whose NodePort 30080 is reachable as localhost:8080
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
nodes:
  - role: control-plane
    extraPortMappings:
      - containerPort: 30080
        hostPort: 8080
```

```
$ kind create cluster --name motd --config kind-config.yaml
$ kubectl cluster-info --context kind-motd
```

The port mapping is how you will reach the service from the browser: the cluster's node port 30080 appears on your machine as port 8080.

Why is the cluster not created by the configuration? It could be, with a provider for `kind`. But a provider has to talk to the cluster when planning the objects inside it, so the cluster has to exist before the plan. Things with different lifecycles belong in different configurations, and "the cluster" and "what runs on it" have very different lifecycles. Chapter 5 uses the same split between infrastructure and application.

Modules

Chapter 3 read a module and called it: a directory of configuration with outputs, pulled from GitHub with a `module` block. That module was a function with no parameters. This chapter writes one with parameters, and the analogy carries: variables are its parameters, resources are its body, and outputs are its return values.

The root module in `kubernetes` still calls the sayings module from GitHub, for its finished SQL this time:

```
module "sayings" {
  source = "github.com/BooksByGorgo/opentofu//tasting/examples/motd?ref=main"
}
```

The application module will be called the same way, with a local path as its `source`: `../modules/motd-k8s`, a directory next to `kubernetes`. After adding or changing a `module` block you run `tofu init` again, which is how modules get installed even when they are just a directory away.

Two pieces of syntax make a module flexible without making it complicated, and both appear in the application module. A variable with `default = null` is optional: `null` means "not set", and the module can test for it with `== null`. When an optional value should turn a whole nested block on or off, a `dynamic "NAME"` block generates zero or more `NAME` blocks from a collection: one per element of its `for_each`, each with the body of its `content`, and the element available inside as `NAME.value`. Think of it as a for loop that emits blocks instead of values. With a collection that is empty when the variable is null and has one element otherwise, it is the idiom for an optional block.

The application module

The application module in `modules/motd-k8s` is the bulk of the chapter. Its `variables.tf` is its signature:

```
variable "namespace" {
  type        = string
  description = "Namespace that holds everything."
  default     = "motd"
}

variable "image" {
  type        = string
  description = "Web server image, e.g. motd:abc123."
}

variable "image_pull_secret" {
  type        = string
  description = "Name of a registry pull secret in the namespace, if the image needs one."
  default     = null
}

variable "seed_sql" {
  type        = string
  description = "SQL that creates and fills the sayings table."
}
```

```
variable "db_password" {
  type        = string
  description = "Password for the motd database user."
  sensitive   = true
}

variable "timezone" {
  type        = string
  description = "Time zone the server uses to pick the current saying."
  default     = "America/Los_Angeles"
}

variable "node_port" {
  type        = number
  description = "Publish the web server on this NodePort; null means ClusterIP only."
  default     = null
}
```

Two variables default to `null`, the optional-variable idiom from the modules section. The module will do something different when they are set, and that is the module author's way of offering an optional feature without making every caller think about it. Every variable has a description, because a module's variables are its documentation.

Namespace, secret, config map

`main.tf` opens with the provider requirement and the first three objects:

```
terraform {
  required_providers {
    kubernetes = {
      source = "hashicorp/kubernetes"
    }
  }
}

resource "kubernetes_namespace_v1" "motd" {
  metadata {
    name = var.namespace
  }
}

locals {
  ns = kubernetes_namespace_v1.motd.metadata[0].name
}

resource "kubernetes_secret_v1" "db" {
  metadata {
    name      = "db"
    namespace = local.ns
  }
  data = {
    password = var.db_password
  }
}
```

```
resource "kubernetes_config_map_v1" "seed" {
  metadata {
    name      = "db-seed"
    namespace = local.ns
  }
  data = {
    "001-sayings.sql" = var.seed_sql
  }
}
```

A module declares which providers it uses, with a `source` but usually without a version, and it does not configure them: the `provider` block lives in the root module and is inherited. The root decides which cluster; the module decides what goes in it.

Every Kubernetes object has a `metadata` block, and `metadata[0].name` is how you read a name back out of one, because `metadata` is a list of blocks even though there is only ever one. The `local.ns` shortcut keeps that from being repeated nine times.

The secret holds the password (Kubernetes base64-encodes it; you hand over the plain value), and the config map holds the seed SQL under the file name MySQL expects.

Tip: A config map holds up to one megabyte. The 3600 sayings are about 230 kilobytes, so there is room, but a real fortune file would not fit. Past the limit you bake the seed into an image or load it with a job.

The database

`database.tf` holds the persistent volume claim, the deployment, and the service:

```
resource "kubernetes_persistent_volume_claim_v1" "db" {
  metadata {
    name      = "db-data"
    namespace = local.ns
  }
  spec {
    access_modes = ["ReadWriteOnce"]
    resources {
      requests = {
        storage = "1Gi"
      }
    }
  }
  # local-path storage binds when the first pod uses the claim
  wait_until_bound = false
}
```

Trap: The provider waits for a claim to be bound by default, and the default storage class in kind (and k3s) binds only when a pod first uses the claim. Leave `wait_until_bound` at its default and `apply` hangs until it times out, with the pod that would bind it waiting on the claim. Set it to `false` for storage classes that bind on first consumer.

The deployment is long because Kubernetes deployments are long; the pattern is the same as the Chapter 3 container, translated:

```
resource "kubernetes_deployment_v1" "db" {
```

```
metadata {
  name      = "db"
  namespace = local.ns
}
spec {
  replicas = 1
  strategy {
    type = "Recreate" # one writer for the volume at a time
  }
  selector {
    match_labels = { app = "db" }
  }
  template {
    metadata {
      labels = { app = "db" }
    }
    spec {
      container {
        name  = "mysql"
        image = "mysql:8.4"
        env {
          name  = "MYSQL_RANDOM_ROOT_PASSWORD"
          value = "yes"
        }
        env {
          name  = "MYSQL_DATABASE"
          value = "motd"
        }
        env {
          name  = "MYSQL_USER"
          value = "motd"
        }
        env {
          name = "MYSQL_PASSWORD"
          value_from {
            secret_key_ref {
              name = kubernetes_secret_v1.db.metadata[0].name
              key  = "password"
            }
          }
        }
        port {
          container_port = 3306
        }
        volume_mount {
          name       = "data"
          mount_path = "/var/lib/mysql"
        }
        volume_mount {
          name       = "seed"
          mount_path = "/docker-entrypoint-initdb.d"
          read_only  = true
        }
        readiness_probe {
```

```
            exec {
              command = ["mysqladmin", "ping", "-h", "127.0.0.1", "--silent"]
            }
            period_seconds = 5
          }
        }
        volume {
          name = "data"
          persistent_volume_claim {
            claim_name = kubernetes_persistent_volume_claim_v1.db.metadata[0].name
          }
        }
        volume {
          name = "seed"
          config_map {
            name = kubernetes_config_map_v1.seed.metadata[0].name
          }
        }
      }
    }
  }

  timeouts {
    create = "5m"
  }
}

resource "kubernetes_service_v1" "db" {
  metadata {
    name      = "db"
    namespace = local.ns
  }
  spec {
    selector = { app = "db" }
    port {
      port = 3306
    }
  }
}
```

The password comes from the secret through `value_from`, so it never appears in the deployment. The `readiness_probe` is the Kubernetes form of the health check from Chapter 3, and it does the same job: the provider waits for a deployment's pods to be ready before calling it created, so the readiness probe is what sequences the database before the web server.

`timeouts` sets how long the provider may wait for create, update, or delete before giving up; it looks like a meta-argument, but each provider defines it for its own resource types, so not every resource has one. Loading 3600 rows takes about twenty seconds; five minutes leaves room for a slow laptop.

The service gives the deployment a stable name, `db`, that the web server can use as a host name. Same idea as the network alias in Chapter 3.

The web server

`web.tf`:

```
resource "kubernetes_deployment_v1" "web" {
  metadata {
    name      = "web"
    namespace = local.ns
  }
  spec {
    replicas = 2
    selector {
      match_labels = { app = "web" }
    }
    template {
      metadata {
        labels = { app = "web" }
      }
      spec {
        dynamic "image_pull_secrets" {
          for_each = var.image_pull_secret == null ? [] : [var.image_pull_secret]
          content {
            name = image_pull_secrets.value
          }
        }
        container {
          name  = "web"
          image = var.image
          env {
            name = "DB_PASSWORD"
            value_from {
              secret_key_ref {
                name = kubernetes_secret_v1.db.metadata[0].name
                key  = "password"
              }
            }
          }
          env {
            name  = "DB_DSN" # kubernetes expands $(DB_PASSWORD) at start
            value = "motd:$(DB_PASSWORD)@tcp(db:3306)/motd"
          }
          env {
            name  = "TZ"
            value = var.timezone
          }
          port {
            container_port = 8080
          }
          readiness_probe {
            http_get {
              path = "/"
              port = 8080
            }
            period_seconds = 5
          }
        }
      }
    }
```

```
  }

  timeouts {
    create = "5m"
  }

  depends_on = [kubernetes_service_v1.db]
}

resource "kubernetes_service_v1" "web" {
  metadata {
    name      = "web"
    namespace = local.ns
  }
  spec {
    type     = var.node_port == null ? "ClusterIP" : "NodePort"
    selector = { app = "web" }
    port {
      port        = 8080
      target_port = 8080
      node_port   = var.node_port
    }
  }
}
```

The `dynamic` block from the modules overview does its work here. A nested block like `image_pull_secrets` is either written or not, and you cannot put a conditional around a block, so the collection is empty when the variable is null and has one element otherwise, and the block is emitted only when a secret was given.

Wut: `$(DB_PASSWORD)` is not an OpenTofu interpolation. OpenTofu only cares about `${...}` with a brace; `$(...)` with parentheses passes through untouched, and Kubernetes expands it from an earlier `env` entry when the container starts. The password therefore goes from secret to container without ever being written into the deployment.

The web deployment's readiness probe hits `/`, which answers 503 until the database answers, so the rollout is complete exactly when the service works. The service is `NodePort` when a port was given and `ClusterIP` otherwise, using the conditional expression from Chapter 3.

`outputs.tf` returns the two names the caller might need:

```
output "namespace" {
  value = local.ns
}

output "web_service" {
  description = "Name of the web service inside the namespace."
  value       = kubernetes_service_v1.web.metadata[0].name
}
```

The root module

The root module in `kubernetes` wires it together. Its `versions.tf` is the one from the start of the chapter, with the Kubernetes provider pointed at the `kind-motd` context. Naming the context means the configuration cannot accidentally deploy to whatever cluster `kubectl` last pointed at, which is a thing that happens.

main.tf:

```
module "sayings" {
  source = "github.com/BooksByGorgo/opentofu//tasting/examples/motd?ref=main"
}

locals {
  app_dir = "${path.module}/app"
  app_hash = sha1(join("", [for f in fileset(local.app_dir, "**") :
    filesha1("${local.app_dir}/${f}")
  ]))
  # the tag names the source, so a new build is a new image name
  image = "motd:${substr(local.app_hash, 0, 12)}"
}

resource "docker_image" "web" {
  name = local.image

  build {
    context = local.app_dir
  }
}

# kind cannot pull from the local docker daemon, so copy the image in
resource "terraform_data" "kind_load" {
  triggers_replace = [docker_image.web.image_id]

  provisioner "local-exec" {
    command = "kind load docker-image ${docker_image.web.name} --name motd"
  }
}

resource "random_password" "db" {
  length  = 24
  special = false
}

module "motd" {
  source = "../modules/motd-k8s"

  image       = docker_image.web.name
  seed_sql    = module.sayings.sql
  db_password = random_password.db.result
  timezone    = var.timezone
  node_port   = 30080

  depends_on = [terraform_data.kind_load]
}
```

The image tag is now the first twelve characters of the source hash instead of `ch2`. That small change does a lot: the image name changes when the source changes, so the deployment's `image` argument changes, so Kubernetes rolls out the new version. There is no `triggers` map any more, because a new name is a new resource. A tag that names the content is the deployment equivalent of a content hash, and it is worth copying.

The `kind load` step is a provisioner, from Chapter 1, doing what provisioners are good for: a command with no provider, run when its inputs change. The `depends_on` on the module makes sure the image is in the cluster before any pod tries to use it. Module arguments are the module's variables; anything without a default must be given.

The check and the outputs are the same as Chapter 3.

Running it

From the `kubernetes` directory:

```
$ tofu init
...
$ tofu apply
docker_image.web: Creation complete after 1s [id=sha256:9c956a29...motd:97bf81ab0552]
terraform_data.kind_load (local-exec): Executing: ["/bin/sh" "-c" "kind load ..."]
terraform_data.kind_load: Creation complete after 2s [id=7a316dca-9679-...]
module.motd.kubernetes_namespace_v1.motd: Creation complete after 0s [id=motd]
module.motd.kubernetes_secret_v1.db: Creation complete after 0s [id=motd/db]
module.motd.kubernetes_persistent_volume_claim_v1.db: Creation complete [id=motd/db-data]
module.motd.kubernetes_config_map_v1.seed: Creation complete after 0s [id=motd/db-seed]
module.motd.kubernetes_service_v1.db: Creation complete after 0s [id=motd/db]
module.motd.kubernetes_service_v1.web: Creation complete after 0s [id=motd/web]
module.motd.kubernetes_deployment_v1.db: Creation complete after 16s [id=motd/db]
module.motd.kubernetes_deployment_v1.web: Creation complete after 26s [id=motd/web]

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

Resources inside a module are addressed as `module.NAME.TYPE.NAME`, and that is how they appear in plans, in `tofu state list`, and in `-replace` arguments.

```
$ curl localhost:8080/
$ kubectl -n motd get pods
[09:43:37] A stubborn race condition expects the unexpected input.
NAME                  READY   STATUS    RESTARTS   AGE
db-85d5847c8c-5qgcc   1/1     Running   0          28s
web-5dc496c778-7jcnm  1/1     Running   0          28s
web-5dc496c778-vvk2x  1/1     Running   0          28s
```

Change the format string in `main.go` and plan:

```
$ tofu plan
  # docker_image.web must be replaced
  # terraform_data.kind_load must be replaced
  # module.motd.kubernetes_deployment_v1.web will be updated in-place
      ~ image = "motd:97bf81ab0552" -> "motd:b860b2aaad47"

Plan: 2 to add, 1 to change, 2 to destroy.
```

A new image, loaded into the cluster, and a rolling update of the web deployment; the database is untouched. Apply, and Kubernetes replaces the two web pods one at a time while the service keeps answering.

Trap: `kubernetes_deployment_v1` waits for the rollout to finish, so a pod that never becomes ready makes apply sit there and then fail with `timed out waiting for the condition`. The reason is never in OpenTofu's output. Run `kubectl -n motd describe pod` and `kubectl -n motd logs` while it waits: an `ImagePullBackOff` means the tag was never loaded into the cluster, and a failing readiness probe usually means the database is not reachable under the name the DSN uses.

Trap: If Docker was installed as a snap on Ubuntu, `kind load` can fail with `permission denied` on a temporary file, because the snap cannot see `/tmp` or hidden directories in your home. Run with `TMPDIR=$HOME/tmp` (any visible directory in your home works) and it succeeds. Docker Desktop on macOS and Windows does not have this problem.

Key Points

- The cluster and the application have different lifecycles and belong in different configurations.
- A module is a function: variables in, resources in the middle, outputs out. The root module configures providers; child modules only declare that they need them.
- `null` defaults make module features optional, and `dynamic` blocks turn an optional value into an optional nested block.
- Readiness probes are how Kubernetes says "ready", and the provider waits for them, so they also sequence your resources.
- A content-addressed image tag turns a rebuild into a visible change in the deployment, which is what triggers a rollout.
- Resources inside modules have addresses like `module.motd.kubernetes_deployment_v1.web`.

New Syntax

Syntax	What it is
`module "NAME" { source = "..." ARGS }`	Call a module; arguments are its variables
`module.NAME.OUTPUT`	Read a module output
`output "x" { precondition { } }`	An assumption checked on an output
`default = null`	An optional variable that is "not set"
`dynamic "BLOCK" { for_each, content { } }`	Generate zero or more nested blocks
`BLOCK.value`	The current element inside a dynamic block
`metadata[0].name`	Reading an attribute of a single-block list
`timeouts { create = "5m" }`	How long a provider may wait
`substr(s, offset, length)`	Take part of a string
`depends_on` on a `module` block	Order a whole module after something

Try It

- Add a `replicas` variable to the module and scale the web deployment from the root without touching the module's resources.
- Install the NGINX ingress controller into kind and add a `kubernetes_ingress_v1` to the module, controlled by an optional `ingress_host` variable and a `dynamic` block.
- Replace the database deployment with a `kubernetes_stateful_set_v1` and see what changes about the volume claim.
- Call the module twice from one root with different namespaces, and find out what breaks first. Then fix it.

- Write a `tofu test` file for the sayings module that asserts `length(output.sayings) == 3600` (look up the `run` block).
- Delete the web pods with `kubectl` and watch Kubernetes recreate them without any help from the configuration. Then run `tofu plan` and see whether OpenTofu noticed.

Exercises

1. **Think about it:** The image tag is derived from the source hash instead of using `motd:latest` with `image_pull_policy = "Always"`. Give two things that would go wrong with the `latest` approach in this configuration.

2. **What does this do?** How many `ports` blocks does this produce when `var.ports = [80, 443]`, and what does `ports.value` refer to in each?

   ```
   dynamic "ports" {
     for_each = var.ports
     content {
       internal = ports.value
       external = ports.value
     }
   }
   ```

3. **Calculation:** The web deployment has `replicas = 2`, and each pod's readiness probe runs every 5 seconds. Roughly how long after the database becomes ready can the deployment be reported created, at the earliest? What in the configuration bounds the worst case?

4. **Where is the bug?**

   ```
   module "motd" {
     source      = "../modules/motd-k8s"
     image       = docker_image.web.name
     db_password = random_password.db.result
     node_port   = 30080
   }
   ```

5. **Where is the bug?**

   ```
   env {
     name  = "DB_DSN"
     value = "motd:${DB_PASSWORD}@tcp(db:3306)/motd"
   }
   ```

6. **Write a module** around `kubernetes_resource_quota_v1`: name it `namespace-quota`, give it a namespace name and a maximum number of pods as variables, and have it create the namespace with that quota. Call it from the root for two namespaces and confirm `kubectl describe quota` in each.

Chapter 5

On the Internet

The need. When a system spans several vendors and several lifecycles, one configuration is no longer the correct unit. Infrastructure that changes rarely and an application that changes daily should be applied separately, with a way to pass values from one to the other. Values that should not be hard-coded, like the newest machine image, have to be looked up. Repeated resources need to be generated from a set rather than written out, and a configuration that reaches across a machine boundary has to be clear about where paths, commands, and secrets end up.

Why it matters. Separate root modules with remote state are how real projects limit the blast radius of a mistake and let people work in parallel. Data sources and `for_each` are what keep a configuration correct as images are retired and lists grow. Knowing where a provider acts, and what loses its sensitivity on the way, is the difference between a secret and a leak.

Why it is hard. A provider must be able to reach its target during plan, so anything the provider needs has to exist before the configuration that uses it, which forces the split. Cloud networks are many small resources that must agree, some of which already exist and have to be taken over rather than created. Credentials for three vendors have to stay out of the files, certificates expire, and destroying things in the wrong order strands a state.

The strategy. Two root modules: infra creates the machine, the DNS records, and the certificate; app reads infra's outputs through `terraform_remote_state` and deploys the Chapter 3 configuration as a module over Secure Shell (SSH). Data sources find the image, `for_each` and `dynamic` generate the records and the firewall rules, a heredoc holds the proxy configuration, and `sensitive()` protects what crosses between states. The example is the sayings server on an Oracle Cloud free tier machine, named through Cloudflare DNS and served over HTTPS with a Let's Encrypt certificate.

The providers

This chapter's two configurations live in `oracle/infra` and `oracle/app`, next to `modules`. Stage one, in `infra`, uses four providers, all new; stage two, in `app`, reuses Docker, random, and HTTP from Chapter 3. `oracle/infra/versions.tf`:

```
terraform {
  required_version = ">= 1.6"

  required_providers {
    oci = {
      source  = "oracle/oci"
      version = "~> 7.0"
    }
```

```
    cloudflare = {
      source  = "cloudflare/cloudflare"
      version = "~> 5.0"
    }
    acme = {
      source  = "vancluever/acme"
      version = "~> 2.0"
    }
    tls = {
      source  = "hashicorp/tls"
      version = "~> 4.0"
    }
  }
}

provider "oci" {
  # reads ~/.oci/config, written by `oci setup config`
}

provider "cloudflare" {
  # reads CLOUDFLARE_API_TOKEN
}

provider "acme" {
  server_url = var.acme_server
}
```

The **OCI** (Oracle Cloud Infrastructure) provider manages Oracle Cloud: networks, subnets, gateways, machines, and hundreds of other resource types, all prefixed `oci_`, plus data sources for looking things up, such as which machine images exist [10]. It authenticates with the same `~/.oci/config` file that Oracle's own command line tool writes and reads. The **Cloudflare** provider manages DNS zones and records, among much else, through Cloudflare's API and an API token [11]. The **ACME** (Automatic Certificate Management Environment) provider speaks the protocol Let's Encrypt uses to issue certificates: `acme_registration` creates an account and `acme_certificate` requests a certificate, proving control of the name through a DNS record that the provider creates and removes for you [12]. The **TLS** (Transport Layer Security) provider generates keys and certificates locally, the way `random` generates strings; this chapter uses `tls_private_key` for the ACME account key [13].

New syntax in this chapter

Five language features appear here for the first time or in a new role. Each is explained where it is used; this is the map.

- **Data sources**, the `data` blocks from Chapter 2's check, now do lookups: which availability domains the region has and which machine image is newest, values you should never hard-code.
- **`for_each`** on a resource creates one instance per element of a set or map, with `each.key` and `each.value` available inside and an address like `RESOURCE["key"]` for each instance; the DNS records use it.
- **`dynamic` over a map** is the Chapter 4 block generator fed a map instead of a list, so that each generated block sees `.key` and `.value`; the firewall rules use it.
- **Heredocs** (`<<-EOT ... EOT`) hold multi-line strings with interpolation, and the `-` strips the common indentation; the proxy configuration is one.
- **`terraform_remote_state`**, a built-in data source, reads another configuration's outputs, and **`sensitive()`** marks a value sensitive again when it lost that on the way; stage two uses both to consume stage one.

Accounts and credentials

Before any configuration, three things must be in place.

Oracle Cloud. Create a free tier account, install the `oci` command line tool, and run `oci setup config`. It writes `~/.oci/config` with your tenancy, user, region, and API key, and the provider reads that file. Find your tenancy OCID (Oracle Cloud Identifier) in the console under your profile; the tenancy is also the root compartment, and building in it is fine for a booklet.

Cloudflare. Your domain's DNS must be hosted by Cloudflare (the free plan is enough). Create an API token with the `Zone.DNS: Edit` permission for the zone, and copy the zone ID from the domain's overview page. Two things read the token, so export it under both names:

```
$ export CLOUDFLARE_API_TOKEN=...       # the cloudflare provider
$ export CLOUDFLARE_DNS_API_TOKEN=...   # the acme provider's dns challenge
```

Let's Encrypt. No account to create; the configuration registers one with an email address. Let's Encrypt has a staging environment for testing and strict rate limits on the real one, and the configuration starts on staging.

Trap: Everything in this chapter that is a secret comes from an environment variable or from `~/.oci/config`. The values that are not secret but are yours — the compartment, the domain, the zone ID — go in `terraform.tfvars`, which the examples' `.gitignore` excludes, with an `example.tfvars` committed to show the form. The one file you never want to see in `git status` is a `.tfvars` with a token in it.

Stage one: the infrastructure

The `infra` directory's providers are the four shown above. The variables are the things that are yours: `compartment_ocid`, `domain`, `cloudflare_zone_id`, `acme_email`, and with defaults `hostnames` (`["motd"]`), `ssh_public_key` (`~/.ssh/id_ed25519.pub`), `admin_cidr` (`0.0.0.0/0`, and you should narrow it), `ad_index` (`0`), and `acme_server` (the staging directory URL). See `variables.tf` in the examples for the descriptions and the validation. If you have no SSH key yet, `ssh-keygen -t ed25519` creates one in that place on macOS, Linux, and Windows alike.

The network

Cloud networking is a small building. The **VCN** (virtual cloud network) is the building. A **subnet** is a floor. The **internet gateway** is the front door, the **route table** is the signs pointing at it, and the **security list** is the guard's list of who may come in. `network.tf`:

```
resource "oci_core_vcn" "motd" {
  compartment_id = var.compartment_ocid
  display_name   = "motd"
  cidr_blocks    = ["10.0.0.0/16"]
  dns_label      = "motd"
}

resource "oci_core_internet_gateway" "motd" {
  compartment_id = var.compartment_ocid
  vcn_id         = oci_core_vcn.motd.id
  display_name   = "motd"
}

# every VCN comes with a default route table; take it over and add a route out
resource "oci_core_default_route_table" "motd" {
```

```
  manage_default_resource_id = oci_core_vcn.motd.default_route_table_id

  route_rules {
    destination       = "0.0.0.0/0"
    network_entity_id = oci_core_internet_gateway.motd.id
  }
}
```

Wut: Creating a VCN silently creates a default route table and a default security list with it. You cannot create them, because they already exist, and you cannot ignore them, because the subnet uses them. The `oci_core_default_*` resource types with `manage_default_resource_id` take over a default object and manage its contents. Several clouds have this pattern; the "default" resource types are the tell.

The security list is where `dynamic` from Chapter 4 pays off, because the rules follow the same pattern three times:

```
locals {
  open_ports = {
    ssh   = { port = 22, source = var.admin_cidr }
    http  = { port = 80, source = "0.0.0.0/0" }
    https = { port = 443, source = "0.0.0.0/0" }
  }
}

resource "oci_core_default_security_list" "motd" {
  manage_default_resource_id = oci_core_vcn.motd.default_security_list_id

  egress_security_rules {
    destination = "0.0.0.0/0"
    protocol    = "all"
  }

  # one ingress rule per entry in local.open_ports
  dynamic "ingress_security_rules" {
    for_each = local.open_ports
    content {
      protocol = "6" # tcp
      source   = ingress_security_rules.value.source
      tcp_options {
        min = ingress_security_rules.value.port
        max = ingress_security_rules.value.port
      }
    }
  }
}

resource "oci_core_subnet" "public" {
  compartment_id = var.compartment_ocid
  vcn_id         = oci_core_vcn.motd.id
  cidr_block     = "10.0.1.0/24"
  display_name   = "public"
  dns_label      = "public"
}
```

`open_ports` is a **map** of **objects**: keys to values, where each value has named fields. Iterating a map with `dynamic` gives `ingress_security_rules.value` (the object) and `ingress_security_rules.key` (the name), and adding a port is adding a line to the map.

The machine

`compute.tf` needs two lookups before it can create anything:

```
locals {
  shape = "VM.Standard.E2.1.Micro" # always free: 1 OCPU, 1 GB
  ads   = data.oci_identity_availability_domains.ads.availability_domains
}

data "oci_identity_availability_domains" "ads" {
  compartment_id = var.compartment_ocid
}

data "oci_core_images" "ubuntu" {
  compartment_id           = var.compartment_ocid
  operating_system         = "Canonical Ubuntu"
  operating_system_version = "24.04"
  shape                    = local.shape
  sort_by                  = "TIMECREATED"
  sort_order               = "DESC"
}
```

Data sources are the read-only lookups from Chapter 2, now doing real work: which availability domains the region has, and which image ids are current Ubuntu for this machine type. The image list is sorted newest first, so `images[0]` is the latest. Hard-coding an image id would break the month the image is retired; the lookup does not.

```
resource "oci_core_instance" "motd" {
  compartment_id      = var.compartment_ocid
  availability_domain = local.ads[var.ad_index].name
  display_name        = "motd"
  shape               = local.shape

  source_details {
    source_type = "image"
    source_id   = data.oci_core_images.ubuntu.images[0].id
  }

  create_vnic_details {
    subnet_id        = oci_core_subnet.public.id
    assign_public_ip = true
    hostname_label   = "motd"
  }

  metadata = {
    ssh_authorized_keys = file(pathexpand(var.ssh_public_key))
    user_data           = base64encode(file("${path.module}/cloud-init.yaml"))
  }
}
```

`metadata` is how the machine is told things at first boot. `ssh_authorized_keys` gets your public key: `pathexpand` turns `~/.ssh/...` into an absolute path and `file` reads it. `user_data` is a cloud-init document,

which Oracle wants base64 encoded, hence `base64encode(file(...))`.

`cloud-init.yaml` prepares the machine so that stage two only needs Docker:

```
#cloud-config
package_update: true
packages:
  - docker.io
runcmd:
  # the Oracle Ubuntu image ships a firewall that only lets SSH in
  - iptables -I INPUT 6 -m state --state NEW -p tcp --dport 80 -j ACCEPT
  - iptables -I INPUT 6 -m state --state NEW -p tcp --dport 443 -j ACCEPT
  - netfilter-persistent save
  # a little swap keeps a 1 GB instance alive while go builds
  - fallocate -l 2G /swapfile
  - chmod 600 /swapfile
  - mkswap /swapfile
  - swapon /swapfile
  - echo '/swapfile none swap sw 0 0' >> /etc/fstab
  # let the ubuntu user drive docker over ssh
  - usermod -aG docker ubuntu
  - systemctl enable --now docker
```

Trap: The security list opens ports 80 and 443 at the network, and then nothing answers. Oracle's Ubuntu images come with `iptables` rules that reject everything except SSH, inside the machine, and the console gives no hint. The two `iptables` lines in cloud-init are the fix, and `netfilter-persistent save` keeps them across reboots. This is the single most common "it does not work" in Oracle's free tier.

Trap: `Out of host capacity` means the availability domain has no free tier machines left, which is common. Try `-var ad_index=1`, or another region if the home region has one domain. The ARM machine type `VM.Standard.A1.Flex` is more generous (four cores and 24 GB free) but far harder to get; if you do get one, Docker builds the image for ARM on the machine itself, so nothing else changes.

The name

`dns.tf` points the names at the machine:

```
locals {
  fqdns = [for h in var.hostnames : "${h}.${var.domain}"]
}

resource "cloudflare_dns_record" "motd" {
  for_each = toset(local.fqdns)

  zone_id = var.cloudflare_zone_id
  name    = each.value
  type    = "A"
  content = oci_core_instance.motd.public_ip
  ttl     = 60
  proxied = false
}
```

`for_each` on a resource creates one instance per element of a set or map. Inside the block, `each.value` is the

element (and `each.key` its key, which for a set is the same thing). `toset` turns the list of names into a set, because `for_each` wants keys that identify each instance, not positions that shift when you insert one in the middle. The instances are addressed as `cloudflare_dns_record.motd["motd.example.com"]`.

`content` is the machine's public address, known only after the instance exists, which orders the record after the machine. `proxied = false` keeps Cloudflare out of the path so that the machine's own certificate is the one browsers see.

The certificate

`cert.tf`:

```
resource "tls_private_key" "account" {
  algorithm   = "ECDSA"
  ecdsa_curve = "P256"
}

resource "acme_registration" "motd" {
  account_key_pem = tls_private_key.account.private_key_pem
  email_address   = var.acme_email
}

resource "acme_certificate" "motd" {
  account_key_pem           = acme_registration.motd.account_key_pem
  common_name               = local.fqdns[0]
  subject_alternative_names = slice(local.fqdns, 1, length(local.fqdns))

  dns_challenge {
    provider = "cloudflare" # reads CLOUDFLARE_DNS_API_TOKEN
  }
}
```

ACME is the protocol Let's Encrypt speaks: prove you control the name, get a certificate for it. The **DNS challenge** proves control by publishing a token in a TXT record, which the provider does through Cloudflare and removes afterwards. It has two advantages over the HTTP challenge: nothing needs to be listening on port 80 yet, and it works for wildcard names.

The first name is the certificate's common name and the rest are alternative names; `slice(list, 1, length)` is "everything but the first".

Tip: Start on the staging server, which the `acme_server` default does. Staging certificates are not trusted by browsers, but they prove the whole chain works, and the production server limits you to five certificates per name per week. Once `https://motd.example.com/` answers (with a warning), apply once more with `-var acme_server=https://acme-v02.api.letsencrypt.org/directory` and the certificate is replaced with a real one.

Wut: A certificate expires after ninety days and the provider renews it by planning a replacement whenever thirty days or fewer remain (`min_days_remaining`). That means renewal is "run `tofu apply` at least monthly", by hand, by cron, or by a scheduled CI job. Nothing renews it while you are not looking.

Outputs

`outputs.tf` is what stage two needs to know:

```
output "public_ip" {
  value = oci_core_instance.motd.public_ip
}

output "fqdn" {
  value = local.fqdns[0]
}

output "certificate_pem" {
  value = acme_certificate.motd.certificate_pem
}

output "issuer_pem" {
  value = acme_certificate.motd.issuer_pem
}

output "private_key_pem" {
  value     = acme_certificate.motd.private_key_pem
  sensitive = true
}
```

Apply stage one with your values in `terraform.tfvars`:

```
$ cd oracle/infra
$ tofu init
$ tofu apply
```

Expect the machine to take a couple of minutes, cloud-init a couple more after that, and the DNS challenge a minute or two; unlike the timings in earlier chapters, these are estimates, since this stage was validated against the provider schemas but not applied while writing the booklet. When it finishes you can log in, and after cloud-init is done, `docker ps` works without `sudo`:

```
$ ssh ubuntu@$(tofu output -raw fqdn) docker ps
```

Stage two: the application

The `app` directory starts by reading stage one's outputs:

```
data "terraform_remote_state" "infra" {
  backend = "local"
  config = {
    path = "${path.module}/../infra/terraform.tfstate"
  }
}

locals {
  infra = data.terraform_remote_state.infra.outputs
}

provider "docker" {
  host     = coalesce(var.docker_host, "ssh://ubuntu@${local.infra.fqdn}")
  ssh_opts = ["-o", "StrictHostKeyChecking=accept-new"]
}
```

`terraform_remote_state` is a data source that reads another configuration's state file and exposes its outputs.

Only outputs: the other configuration decides what it publishes, and the rest of its state stays private. With a `local` backend the config is a path; with a shared backend it is the same settings the other configuration uses (Appendix A).

The Docker provider then points at the machine over SSH. The provider runs the same `ssh` you do (on Windows, the OpenSSH client that ships with the system), so your key and agent are used, and `accept-new` records the host key on first contact instead of stopping to ask, which would hang an apply. The `docker_host` variable defaults to `null` and exists so that the same configuration can be pointed at any Docker host, including your own for a dry run; `coalesce` returns its first argument that is not null.

Trap: The provider has to reach the machine during `plan`, not just `apply`. Run stage two before stage one is done and you get a connection error, not a plan. That is the reason for two stages: a provider's configuration has to be known, and reachable, before planning the things it manages.

Chapter 3 as a module

The Chapter 3 configuration moved into `modules/motd-docker` with four changes, and each teaches something about modules.

Values became variables. `app_dir`, `seed_sql`, `db_password`, `timezone`, and `port` are inputs now; the module builds the same network, volume, database, and web server from them.

The bind mount became an `upload`. Chapter 3 wrote the seed to a file and bind-mounted its directory into the database container. That file is on your laptop, and the Docker host is now a machine in Oracle's data center, where the path does not exist. The `upload` block copies content into the container at creation instead:

```
  upload {
    content = var.seed_sql
    file    = "/docker-entrypoint-initdb.d/001-sayings.sql"
  }
```

Wut: Paths in a `volumes` block are paths on the Docker host, not on the machine running `tofu`. That is invisible when they are the same machine and very visible when they are not. The same goes for `local-exec` provisioners, which run where `tofu` runs, and `remote-exec` provisioners, which run on the resource. Always ask "where does this path live" when a configuration crosses a machine boundary.

A resource stands in for the seed. `replace_triggered_by` only accepts resource references, and there is no `local_file` any more, so the module wraps the SQL in a `terraform_data` and the volume watches that:

```
resource "terraform_data" "seed" {
  input = var.seed_sql
}

resource "docker_volume" "db_data" {
  name = "motd-db-data"

  lifecycle {
    replace_triggered_by = [terraform_data.seed]
  }
}
```

The port became optional. A `dynamic "ports"` block over `var.port == null ? [] : [var.port]` publishes the web server's port only when asked, the Chapter 4 idiom. Stage two does not ask, because only the proxy should talk to the web server.

The module's outputs are the network name and the web container's name, which are the two things the proxy needs.

The proxy

`main.tf` in `app`:

```
module "sayings" {
  source = "github.com/BooksByGorgo/opentofu//tasting/examples/motd?ref=main"
}

resource "random_password" "db" {
  length  = 24
  special = false
}

module "motd" {
  source = "../../modules/motd-docker"

  app_dir     = "${path.module}/app"
  seed_sql    = module.sayings.sql
  db_password = random_password.db.result
  timezone    = var.timezone
  # no host port: only the proxy talks to the web server
}

locals {
  caddyfile = <<-EOT
    ${local.infra.fqdn} {
        tls /certs/tls.crt /certs/tls.key
        reverse_proxy ${module.motd.web_container}:8080
    }
  EOT
}
```

The Caddyfile is a **heredoc**: `<<-EOT` starts a multi-line string that ends at a line containing only `EOT`, and the `-` strips the common leading indentation so that the configuration can stay indented. Interpolation works inside it. Caddy is told the name it serves, where its certificate is, and where to send requests; given a name and a certificate it serves HTTPS on 443 and redirects port 80 by itself.

```
resource "docker_image" "caddy" {
  name = "caddy:2"
}

resource "docker_container" "proxy" {
  name  = "motd-proxy"
  image = docker_image.caddy.image_id

  networks_advanced {
    name = module.motd.network
  }

  ports {
    internal = 80
    external = var.http_port
```

```
  }

  ports {
    internal = 443
    external = var.https_port
  }

  upload {
    content = local.caddyfile
    file    = "/etc/caddy/Caddyfile"
  }

  upload {
    content = "${local.infra.certificate_pem}${local.infra.issuer_pem}"
    file    = "/certs/tls.crt"
  }

  upload {
    content = sensitive(local.infra.private_key_pem)
    file    = "/certs/tls.key"
  }
}
```

Three uploads: the Caddyfile, the certificate with its issuer chain appended, and the private key. A changed upload replaces the container, so a renewed certificate in stage one becomes a new proxy in stage two.

Trap: Outputs lose their sensitivity when read through `terraform_remote_state`. Without the `sensitive()` wrapper, the private key is printed in full in every plan that creates the proxy, and in the logs of whatever CI ran it. Wrap any secret that comes from another state, and check the plan for anything that should say `(sensitive value)` and does not.

The check is the one from Chapter 3 with the real URL and `insecure = true`, because staging certificates are not trusted by the HTTP provider either.

Running it

```
$ cd ../app
$ tofu init
$ tofu apply
```

The first apply builds the image on the machine, which should take a few minutes on one core with a gigabyte of memory (the swap in cloud-init is what makes it possible), pulls MySQL and Caddy, loads the sayings, and starts the proxy. That estimate is untested too: the application stage was applied against a local Docker daemon with a self-signed certificate standing in for stage one, not against the Oracle machine. Then:

```
$ curl -k $(tofu output -raw url)
```

Or open the URL in a browser, accept the staging warning, and read your saying. Switch stage one to the production ACME server, apply it, apply stage two, and the warning is gone.

Trap: Destroy in the reverse order of creation: `app` first, then `infra`. Destroying `infra` first deletes the machine that `app`'s Docker provider needs to reach, and `tofu destroy` in `app` then fails trying to refresh containers that no longer exist. If it happens, `tofu destroy -refresh=false` in `app` or deleting its state gets you unstuck; the containers died with the machine.

Key Points

- Split configurations by lifecycle: infrastructure that changes rarely, application that changes often. `terraform_remote_state` passes outputs from one to the other.
- Cloud networks are buildings: VCN, subnet, gateway, route table, security list. Default objects are managed with the `default` resource types.
- Data sources look up things you should not hard-code, like the newest image.
- cloud-init prepares a machine at first boot, and on Oracle it must also open the machine's own firewall.
- `for_each` makes one resource instance per element of a set or map, addressed by key.
- The ACME DNS challenge issues certificates without needing anything listening, and renewal means applying again once thirty days or fewer remain.
- Paths and commands run where the provider's target is, not where you are; `upload` moves content across that boundary.
- Secrets read from another state are not sensitive until you say so.

New Syntax

Syntax	What it is
`data "TYPE" "NAME" { filters }`	A lookup, with results in `data.TYPE.NAME.ATTR`
`for_each = toset(list)` with `each.value`	One resource instance per element
`RESOURCE["key"]`	Address of one instance of a `for_each` resource
`{ a = { port = 22 } }`	A map of objects
`dynamic` over a map: `.key`, `.value`	Iterating a map into nested blocks
`<<-EOT ... EOT`	Indented heredoc string
`file`, `pathexpand`, `base64encode`, `toset`, `slice`	More built-in functions
`data "terraform_remote_state"`	Read another configuration's outputs
`sensitive(x)`	Mark a value sensitive
`manage_default_resource_id`	Take over a cloud's default object
`upload { content, file }`	Copy content into a container at creation

Try It

- Add `"www"` to `hostnames` and apply both stages. Which resources change in each?
- Remove the certificate plumbing and let Caddy get its own certificate: a Caddyfile with just the name and `reverse_proxy` does it, using the HTTP challenge on port 80. Compare the two approaches: what does each need open, and who renews?
- Move stage one's state to Oracle Object Storage using the `s3` backend with the compatibility endpoint, and change stage two's remote state to match.
- Write a GitHub Actions workflow that runs `tofu apply` on stage one weekly to renew the certificate, with the tokens in repository secrets.
- Get an `A1.Flex` machine, install k3s from cloud-init, fetch its kubeconfig, and deploy the Chapter 4 module to it with a `kubernetes_ingress_v1` that uses the certificate as a `kubernetes.io/tls` secret. You will need to push the image to a registry; `ghcr.io` with a GitHub token that has the `write:packages` scope works.
- Narrow `admin_cidr` to your own address and confirm SSH from elsewhere fails and HTTPS still works.

Exercises

1. **Think about it:** Stage two reads stage one's outputs through remote state. Why not put everything in one configuration, with the Docker provider's `host` built from `oci_core_instance.motd.public_ip`

by interpolation?

2. **What does this do?** With `hostnames = ["motd", "www"]` and `domain = "example.com"`, what are the addresses of the DNS record instances, and what are `common_name` and `subject_alternative_names` on the certificate?

3. **Calculation:** A certificate was issued on March 1st with `min_days_remaining = 30`. On which day does `tofu plan` first show it being replaced, and if you apply only on the first of each month, how many days of validity does the old certificate have left on the day it is replaced?

4. **Where is the bug?**

```
resource "cloudflare_dns_record" "motd" {
  for_each = local.fqdns

  zone_id = var.cloudflare_zone_id
  name    = each.value
  type    = "A"
  content = oci_core_instance.motd.public_ip
  ttl     = 60
}
```

5. **Where is the bug?** The machine is up, the security list opens 443, DNS resolves, Caddy is running, and `curl https://motd.example.com/` times out.

6. **Write a configuration** for a second, staging machine in the same VCN with its own name (`motd-staging.example.com`), sharing the subnet and security list, by turning the instance, the record, and the certificate into a module called twice. Decide what the module's variables should be before you write it.

Chapter 6

Conclusion

You started with a configuration that ran `echo` and ended with a service on the internet, with a name, a certificate, a database, and a rollout strategy, all described in files that a colleague could apply from nothing. Here are the key takeaways:

- **Describe the end state, not the steps.** OpenTofu computes the steps, shows them as a plan, and applies only the difference. Applying twice is safe.
- **State is memory.** It maps your blocks to real things, it holds secrets, and it never goes into git.
- **Providers are the vocabulary.** Every resource type comes from one, credentials come from the environment, and the lock file keeps everyone on the same versions.
- **References are dependencies.** Order comes from what mentions what, and `depends_on` is for the rare case with nothing to mention.
- **Validate in layers.** `fmt`, `validate`, `validation` blocks, `precondition`, the plan, and `check` blocks, each catching what the one before could not.
- **Make change visible.** Content hashes and content-addressed tags turn "the source changed" into a plan line.
- **Ready means ready.** Health checks and readiness probes make "created" mean "serving", and everything downstream gets sequenced for free.
- **Modules are functions.** Variables in, resources in the middle, outputs out, and a module with no resources is still a useful function.
- **Split by lifecycle.** Clusters and machines in one configuration, applications in another, with remote state carrying the outputs across.
- **Know where things run.** Paths, commands, and bind mounts live on the provider's target, and `upload` carries content across the boundary.
- **Secrets are only as sensitive as you say.** Mark them, wrap what crosses states in `sensitive()`, and protect the state that holds them.

Everything you built here can be torn down with `tofu destroy`, in the reverse order it was built, and rebuilt with `tofu apply`. That round trip is the whole promise of infrastructure as code, and now you have made it work five times. Go describe something.

Content outline and editorial support from Ben. Words by Claude.

Appendix A

Best Practices, Recommendations, and Common Errors

This appendix is the booklet's reference card. The chapters introduce these ideas where they come up; this is where they are collected, with the reasoning kept short and a pointer to the documentation for each item. Everything here applies to OpenTofu and Terraform alike unless a line says otherwise, and the two projects' documentation sites cover the same ground in the same words most of the time [14], [15].

Best practices

Organizing configuration

- **One directory per root module, one state per root module.** A root module is a unit of `apply`. Things with different lifecycles (a cluster and what runs on it, infrastructure and application) go in different root modules, as Chapters 4 and 5 do [16], [17].
- **Use the conventional file names.** `versions.tf` (the `terraform` block), `providers.tf`, `variables.tf`, `outputs.tf`, `main.tf`, and more files named for what they hold (`network.tf`, `dns.tf`). OpenTofu does not care; the next reader does [17], [18].
- **Name resources for their role, not their type:** `web`, not `web_container`. The type is already in the address. Use underscores in names, never dashes, and keep names short: they appear in every plan [17], [19].
- **Keep modules small and single purpose,** with every variable described and every output named for what it is. A module is a function; write it like one [18].
- **Do not abstract too early.** Three similar resources are fine. The third copy of a whole configuration is the moment to write a module [20].
- **Give every variable a `type` and a `description`,** and a `validation` block when a wrong value would fail late or expensively [21].
- **Prefer references to `depends_on`.** A reference is a dependency the reader can see. Use `depends_on` only when nothing can be referenced, and comment why [22].

Versions and providers

- **Constrain the provider major version** with `~>` in `required_providers`, and set `required_version` to the oldest tool version you tested [23], [24].
- **Commit `.terraform.lock.hcl`.** It pins exact provider versions and checksums so that everyone gets the same plugins. When the team spans platforms, run `tofu providers lock -platform=linux_amd64 -platform=darwin_arm64` and commit the result, or `init` on the other platform fails with a checksum

error [25], [26]. (The examples in this booklet ignore the lock file for that reason; a real project should not.)
- **Upgrade providers deliberately** with `tofu init -upgrade`, read the changelog first, and commit the lock file change on its own [27].
- **Never commit `.terraform/`.** It is a cache [25].

State

- **Never commit state.** It holds secrets and it changes on every apply. `.gitignore` gets `*.tfstate`, `*.tfstate.*`, `.terraform/`, and `crash.log` before the first apply [28].
- **Use a remote backend with locking** as soon as two people or one CI job touch a configuration: S3 and compatible stores (Oracle Object Storage among them), Azure Blob, Google Cloud Storage, a Kubernetes secret, or an HTTP backend. Locking prevents two applies from interleaving; encryption at rest covers the secrets [29], [30], [31].
- **Never edit state by hand.** `tofu state list`, `show`, `mv`, `rm`, and `tofu import` cover what is needed, and `moved`, `removed`, and `import` blocks do the same declaratively and reviewably [32], [33], [34].
- **Back up state** before anything unusual, with `tofu state pull > backup.tfstate` [35].
- **One state per environment,** by directory or by backend key, not by `workspace` unless the differences are truly only variable values [36].

Planning and applying

- **Always read the plan.** The summary line first, then every `-`/`+`. A replacement of anything stateful is a stop-and-think moment [37].
- **Save the plan in automation:** `tofu plan -out=tfplan` then `tofu apply tfplan`. A saved plan applies exactly what was reviewed, and applying a stale one fails instead of surprising you [37], [38].
- **Run `fmt`, `validate`, and `plan` in CI on every pull request,** and apply from CI on merge, so that state changes are serialized and logged [39], [40].
- **Avoid `-target`.** It is for surgery, not routine. A configuration that needs `-target` to apply cleanly has a dependency problem to fix [37].
- **Do not `-auto-approve` anything that matters.** Interactively, read and type `yes`; in CI, apply a saved plan that was reviewed [38].
- **Treat `destroy` as a real command.** Protect stateful resources with `lifecycle { prevent_destroy = true }` and remove the protection only in the same change that intends the destruction [5], [41].

Secrets

- **Credentials come from the environment or the vendor's own config files,** never from `.tf` or committed `.tfvars` files. `TF_VAR_name` sets a variable from the environment [42], [43].
- **Mark secret variables and outputs `sensitive`,** and wrap values read from another configuration's outputs in `sensitive()` [21], [44].
- **Remember that state holds the plain value regardless.** Sensitivity is about the terminal and the logs; the backend's access control is about the state [28].
- **Generate secrets with the `random` provider or fetch them from a secret manager** rather than typing them, so that they are never in a shell history [45].

Changing infrastructure

- **Prefer immutable artifacts.** An image tagged with its content hash (Chapter 4) is a change OpenTofu can see; `latest` is not [46], [47].
- **Provisioners are a last resort.** They run only at creation, their failures leave resources tainted, and their commands are invisible to the plan. Prefer a provider, cloud-init, or a proper configuration management tool, and use `terraform_data` with `local-exec` for the small glue that has no better home [48], [49].

- **Make replacement chains explicit** with `replace_triggered_by` when the provider cannot see the dependency (a seed file and the volume it loads into) [5].
- **Add `precondition` and `postcondition` blocks** for assumptions that would otherwise fail late, and `check` blocks for "did it work" [6], [50].
- **Refactor with `moved` blocks,** not by destroying and recreating. Renaming a resource or moving it into a module is a `moved { from = ...; to = ... }` and a plan that says "no changes" [33].

Recommendations

Tools

- `tofu fmt -recursive` and `tofu validate` in a pre-commit hook [51], [52], [53].
- **tflint** for lint rules OpenTofu does not enforce (unused variables, deprecated arguments, provider-specific checks) [54].
- **trivy** or **checkov** for security scanning of configurations (open security groups, unencrypted storage) [55], [56].
- **terraform-docs** to generate the variables and outputs tables of a module's README from its files [57].
- **tenv** (or a similar version manager) to install and switch between OpenTofu and Terraform versions per project [58].
- An editor with the language server, which gives completion for every provider's arguments [59].

Working with OpenTofu

- `tofu console` to try an expression before putting it in a file [60].
- `tofu show` to read the state or a saved plan; `tofu show -json tfplan` for scripts [61].
- `tofu graph | dot -Tsvg > graph.svg` to see the dependency graph when the order surprises you [62].
- `tofu plan -refresh-only` to see drift (changes made outside OpenTofu) without planning any fixes, and `tofu apply -refresh-only` to accept it into the state [37], [63].
- `tofu apply -replace=ADDRESS` to force one resource to be recreated, instead of tainting or deleting things [37].
- `TF_LOG=DEBUG tofu plan` when a provider does something inexplicable; it prints every API call [64].
- `tofu test` with `.tftest.hcl` files for modules that other configurations depend on [65].
- `tofu providers schema -json` to see every argument a provider accepts when the documentation and reality disagree [66].

Staying compatible with both tools

Everything in this booklet works in both. Features to avoid, or to fence off, if a configuration must run under either [67]:

Only in OpenTofu	Only in Terraform
State encryption (`encryption` block) [68]	HCP Terraform and `cloud` block features [69]
Variables in `backend` and module `source` [70]	Terraform Stacks [71]
`for_each` on `provider` blocks [72]	`ephemeral` resources and write-only arguments [73], [74]
`-exclude` flag [72]	`terraform query` and list resources [75]
`.tofu` file extension [70]	Some newer built-in functions [15]
Provider mirrors from OCI registries [76], [77]	

Both have `check` blocks, `import`, `moved`, and `removed` blocks, `terraform_data`, provider-defined functions, and `test` [78]. The version numbers do not line up: OpenTofu started at 1.6 (equivalent to Terraform 1.5), so `required_version = ">= 1.6"` means something slightly different under each tool [67].

Common errors

The message, what it usually means, and what to do. Messages are abbreviated.

Setup

Message	Cause and fix
`Required plugins are not installed` or `Missing required provider`	Run `tofu init` [27].
`Inconsistent dependency lock file`	The lock file and the constraints disagree; `tofu init` if a constraint changed, `tofu init -upgrade` to move to newer versions [25], [27].
`Backend initialization required`	The backend settings changed; `tofu init -migrate-state` to move state, `-reconfigure` to point at existing state [27], [31].
`Failed to query available provider packages`	Wrong `source` address, a typo in the version, or no network. Check the registry page for the exact address [24].
`Unsupported OpenTofu Core version`	`required_version` excludes the version you are running; upgrade OpenTofu or loosen the constraint [79].
`Error acquiring the state lock`	Another apply is running, or one crashed and left the lock; wait, then `tofu force-unlock ID` only when you are sure it is stale [30], [80].

Configuration

Message	Cause and fix
`Unsupported argument` (with "Did you mean")	A misspelled argument, or an argument from a different provider version [52].
`Missing required argument`	The resource type needs it; the docs list which are required [81].
`Reference to undeclared resource` or `undeclared input variable`	A typo in a reference, or a missing `variable` block. Remember `var.`, `local.`, `data.`, `module.` prefixes [82].
`Invalid function argument` or `Invalid template interpolation value`	A type mismatch, often a number where a string is needed; use `tostring`, `tonumber`, or check the type in `tofu console` [83], [84].
`Invalid for_each argument ... known after apply`	`for_each` keys must be known at plan time; use static keys, derive them from configuration rather than from another resource's attributes, or split into two applies [85].
`Cycle:` followed by addresses	Resources reference each other in a loop; break it with a data source, a local, or by removing an unneeded `depends_on` [22], [86].
`Duplicate resource`	Two blocks with the same type and name, often in two files [81].
`Invalid value for variable`	A `validation` block rejected the value; the message is whatever the author wrote [21].
`Resource precondition failed`	A `precondition` was false; fix the assumption or the configuration [6].

Planning and applying

Message	Cause and fix
`Saved plan is stale`	State changed since the plan was saved; plan again [38].
`Objects have changed outside of OpenTofu`	Drift. Read what changed; `apply -refresh-only` to accept it, or a normal apply to revert it [37], [63].
`Provider produced inconsistent final plan`	A provider bug or a value that changed between plan and apply; re-run, then report it if it repeats [64].
`Instance cannot be destroyed`	`prevent_destroy` is doing its job; remove it deliberately if you mean it [5].
`local-exec provisioner error`	The command failed; the resource is tainted and the next apply recreates it. Fix the command [48].
`Check block assertion failed`	A warning: the apply succeeded but the check did not. Look at what the check tests [50].
`Provider configuration not present`	A resource in state belongs to a provider you removed from the configuration; add the provider back long enough to destroy it, or `tofu state rm` it [43].
`Error: ... already exists` or HTTP 409	The thing exists but not in state; `tofu import` it or an `import` block, or remove the stray one [34], [87].

Providers used in this booklet

Message	Cause and fix
`Bind for 0.0.0.0:8080 failed: port is already allocated`	Another process or container owns the port; change the variable [88].
`Conflict. The container name "/motd-db" is already in use`	A container OpenTofu does not know about; remove it or import it [87], [89].
`Cannot connect to the Docker daemon`	Docker is not running, or the socket needs your user in the `docker` group, or the `host` is wrong [90], [91].
`timed out waiting for the condition` (Kubernetes)	The rollout never became ready; `kubectl describe pod` and `kubectl logs` say why [9], [92].
`ImagePullBackOff`	The cluster cannot pull the image: not loaded into kind, not pushed to the registry, or no pull secret [46], [93].
`Unauthorized` (Kubernetes)	The kubeconfig context is wrong or expired [94].
`Out of host capacity` (Oracle)	No free tier machines in that availability domain; try another `ad_index` or region [95], [96].
`NotAuthorizedOrNotFound` (Oracle)	The compartment OCID is wrong, or the profile in `~/.oci/config` lacks permission, or the region differs from the resource's [10], [97].
`Authentication error (10000)` (Cloudflare)	The token is missing, wrong, or lacks `Zone.DNS: Edit` on that zone [11], [98].
`urn:ietf:params:acme:error:rateLimited`	Too many certificates for the name; wait, and use the staging server while testing [99], [100], [101].
`NXDOMAIN` during the DNS challenge	The TXT record has not propagated; the provider retries, and a low TTL (time to live) on the zone helps [102], [103].
`Permission denied (publickey)` over SSH	The key in `ssh_public_key` is not the one your agent offers, or cloud-init has not finished creating the user [104], [105].

Appendix B

Built-ins

Everything in this booklet came from one of two places: a provider, or OpenTofu itself. This appendix lists what OpenTofu itself provides, with no provider and no download: one resource type and one data source, the block types, the meta-arguments every resource accepts, the named values you can refer to, the operators, and the built-in functions. The function list is the commonly used subset; the full list is longer and lives in the documentation [106]. Every example below was evaluated with `tofu console`, and the results are shown as plain values; the console itself wraps typed collections as `tolist([...])`, `toset([...])`, or `tomap({...})`.

The built-in provider

The built-in provider needs no `required_providers` entry and defines two things [107]:

Name	What it is	Chapter
`terraform_data` resource	A placeholder resource. Arguments: `input` (any value, echoed back as the `output` attribute) and `triggers_replace` (a list; when any element changes, the resource is replaced). Attributes: `id`, `output` [49].	1
`terraform_remote_state` data source	Reads another configuration's state. Arguments: `backend` and `config` (the other configuration's backend settings), optional `workspace`. Attribute: `outputs`, a map of that configuration's outputs [108].	5

Block types

The top-level blocks a configuration is made of, and the nested blocks that only appear inside them:

Block	Purpose	Chapter
`terraform { }`	Settings for OpenTofu: `required_version`, `required_providers`, `backend` [79]	1, 2
`provider "NAME" { }`	Configure a provider [43]	2
`resource "TYPE" "NAME" { }`	A thing that should exist [81]	1
`data "TYPE" "NAME" { }`	A read-only lookup [109]	2
`variable "NAME" { }`	An input: `type`, `default`, `description`, `sensitive`, `validation` [21]	1

Block	Purpose	Chapter
`output "NAME" { }`	A return value: `value`, `description`, `sensitive`, `precondition` [110]	1
`locals { }`	Named expressions [111]	2
`module "NAME" { }`	Call a child module: `source`, `version`, its variables [112]	3
`check "NAME" { }`	A test that runs after plan and apply, holding `data` blocks and `assert` blocks [50]	2
`import { }`	Bring an existing object into state: `to`, `id` [34]	A
`moved { }`	Record a rename: `from`, `to` [33]	A
`removed { }`	Forget a resource without destroying it [81]	A
`provisioner "TYPE" { }`	Inside a resource: a command at creation (`local-exec`, `remote-exec`, `file`) [48]	1
`lifecycle { }`	Inside a resource: the meta-arguments below [5]	3
`dynamic "BLOCK" { }`	Inside a resource: generate nested blocks from a collection	4

Meta-arguments

Arguments OpenTofu understands on every `resource`, `data`, and (where noted) `module` block, regardless of provider:

Meta-argument	Meaning
`depends_on = [ ... ]`	Create after these, destroy before them; also on modules [22]
`count = N`	Create N instances, addressed `NAME[0]` and so on, with `count.index` inside [113]
`for_each = set or map`	One instance per element, addressed `NAME["key"]`, with `each.key` and `each.value` inside; also on modules [85]
`provider = NAME.ALIAS`	Use a non-default provider configuration [114]
`lifecycle { create_before_destroy = true }`	On replacement, create the new one first
`lifecycle { prevent_destroy = true }`	Fail any plan that would destroy this
`lifecycle { ignore_changes = [ ... ] }`	Do not plan updates for these arguments (drift you accept)
`lifecycle { replace_triggered_by = [ ... ] }`	Replace this when those resources change
`lifecycle { precondition { } }`	An assumption checked before the resource is planned
`lifecycle { postcondition { } }`	A guarantee checked after the resource is applied, with `self` available

`timeouts { }` looks like a meta-argument but is defined per resource type by its provider, so not every resource has one.

Named values

What an expression can refer to [82]:

Reference	What it is
`var.NAME`	An input variable
`local.NAME`	A local value
`TYPE.NAME` and `TYPE.NAME.ATTR`	A resource and its attributes
`data.TYPE.NAME.ATTR`	A data source result
`module.NAME.OUTPUT`	A child module's output
`self.ATTR`	The current resource, inside provisioners and postconditions
`each.key`, `each.value`	The current element inside a `for_each` resource or dynamic block
`count.index`	The current index inside a `count` resource
`path.module`, `path.root`, `path.cwd`	The module's directory, the root module's directory, the shell's directory
`terraform.workspace`	The current workspace name (`default` unless you use workspaces)

Operators and expressions

Syntax	Meaning
`+ - * / %`	Arithmetic; `/` is floating point, `%` is remainder [115]
`== != < <= > >=`	Comparison
`&& || !`	Logical and, or, not
`cond ? a : b`	Conditional [116]
`[for x in list : expr]`	For expression producing a list [117]
`{for k, v in map : k => expr}`	For expression producing a map
`[for x in list : expr if cond]`	For expression with a filter
`list[*].attr`	Splat: the attribute of every element [118]
`"a ${expr} b"`	String interpolation [119]
`"%{ if cond }yes%{ endif }"`	String directive; also `%{ for }`
`<<-EOT ... EOT`	Indented heredoc
`f(a, b)` and `f(list...)`	Function call; `...` spreads a list into arguments [120]
`list[0]`, `map["key"]`, `map.key`	Indexing and attribute access

Functions

Strings

Example	Result
`format("%s-%03d", "web", 7)`	`"web-007"`
`join("-", ["a", "b", "c"])`	`"a-b-c"`
`split(",", "a,b,c")`	`["a", "b", "c"]`
`replace("hello world", "world", "gorgo")`	`"hello gorgo"`
`trimspace("  hi  ")`	`"hi"`
`trim("xxhixx", "x")`	`"hi"`
`trimprefix("motd:ch2", "motd:")`	`"ch2"`
`trimsuffix("main.go", ".go")`	`"main"`
`lower("Hello")`, `upper("hello")`	`"hello"`, `"HELLO"`
`title("hello world")`	`"Hello World"`
`substr("abcdef", 2, 3)`	`"cde"`

Example	Result
`startswith("hello", "he")`	`true`
`endswith("main.go", ".go")`	`true`
`strcontains("hello", "ell")`	`true`
`regex("[0-9]+", "port 8080")`	`"8080"` (an error when nothing matches)
`regexall("[a-z]+", "a1b2")`	`["a", "b"]`
`indent(2, "a\nb")`	`"a\n  b"` (every line but the first)
`chomp("line\n")`	`"line"`
`length("hello")`	`5`

Collections

Example	Result
`length(["a", "b", "c"])`	`3`
`element(["a", "b", "c"], 4)`	`"b"` (the index wraps around)
`index(["a", "b", "c"], "b")`	`1`
`lookup({a = 1}, "b", 0)`	`0` (the default)
`keys({a = 1, b = 2})`	`["a", "b"]`
`values({a = 1, b = 2})`	`[1, 2]`
`merge({a = 1}, {b = 2})`	`{a = 1, b = 2}`
`concat([1, 2], [3])`	`[1, 2, 3]`
`flatten([[1, 2], [3]])`	`[1, 2, 3]`
`distinct([1, 1, 2])`	`[1, 2]`
`sort(["b", "a"])`	`["a", "b"]`
`reverse([1, 2, 3])`	`[3, 2, 1]`
`slice([1, 2, 3, 4], 1, 3)`	`[2, 3]`
`contains(["a", "b"], "a")`	`true`
`range(3)`	`[0, 1, 2]`
`zipmap(["a", "b"], [1, 2])`	`{a = 1, b = 2}`
`setproduct(["a", "b"], [1, 2])`	`[["a", 1], ["a", 2], ["b", 1], ["b", 2]]`
`setunion([1], [2])`	`[1, 2]` (as a set)
`one([42])`	`42` (an error for more than one element)
`coalesce("", "x")`	`"x"` (the first non-empty)
`coalescelist([], [1])`	`[1]`
`compact(["a", "", "b"])`	`["a", "b"]`
`alltrue([true, false])`	`false`
`anytrue([true, false])`	`true`
`sum([1, 2, 3])`	`6`

Numbers

Example	Result
`min(3, 1, 2), max(3, 1, 2)`	`1, 3`
`abs(-4)`	`4`
`ceil(1.2), floor(1.8)`	`2, 1`
`pow(2, 10)`	`1024`
`parseint("ff", 16)`	`255`

Example	Result

Types and errors

Example	Result
`tostring(42)`	`"42"`
`tonumber("42")`	`42`
`tobool("true")`	`true`
`tolist(toset(["b", "a", "b"]))`	`["a", "b"]`
`toset(["b", "a", "b"])`	`["a", "b"]` (as a set)
`tomap({a = 1})`	`{a = 1}`
`type(["a"])`	`tuple([string])`
`can(regex("[0-9]+", "abc"))`	`false` (the error became false)
`try(regex("[0-9]+", "abc"), "none")`	`"none"` (the first argument that does not error)
`sensitive("secret")`	`(sensitive value)`
`nonsensitive(sensitive("secret"))`	`"secret"`

Encoding

Example	Result
`jsonencode({a = 1})`	`"{\"a\":1}"`
`jsondecode("{\"a\": 1}")`	`{a = 1}`
`yamlencode({a = [1, 2]})`	`"\"a\":\n- 1\n- 2\n"`
`yamldecode("a: 1")`	`{a = 1}`
`base64encode("hi")`	`"aGk="`
`base64decode("aGk=")`	`"hi"`
`urlencode("a b&c")`	`"a+b%26c"`

Files and paths

Example	Result
`file("hello.txt")`	The file's contents as a string
`fileexists("hello.txt")`	`true`
`fileset("app", "*.go")`	`["main.go"]` (as a set)
`filesha1("hello.txt")`	`"f572d396..."`
`filesha256("hello.txt")`	`"5891b5b5..."`
`templatefile("greet.tftpl", { name = "Gorgo" })`	`"Hi Gorgo!\n"` for a template holding `Hi ${name}!`
`abspath("app")`	`"/home/you/webserver/app"`
`dirname("app/main.go")`	`"app"`
`basename("app/main.go")`	`"main.go"`
`pathexpand("~/.kube/config")`	`"/home/you/.kube/config"`

Hashes and identifiers

Example	Result
`sha1("hello")`	`"aaf4c61d..."`
`sha256("hello")`	`"2cf24dba..."`
`md5("hello")`	`"5d41402a..."`
`uuid()`	A new random universally unique identifier (UUID) on every evaluation

Dates

Example	Result
`timestamp()`	The current UTC time, RFC 3339, new on every evaluation
`formatdate("YYYY-MM-DD", "2026-09-05T10:30:00Z")`	`"2026-09-05"`
`timeadd("2026-09-05T10:30:00Z", "48h")`	`"2026-09-07T10:30:00Z"`

Networks

Example	Result
`cidrsubnet("10.0.0.0/16", 8, 1)`	`"10.0.1.0/24"`
`cidrhost("10.0.1.0/24", 5)`	`"10.0.1.5"`
`cidrnetmask("10.0.1.0/24")`	`"255.255.255.0"`
`cidrsubnets("10.0.0.0/16", 8, 8)`	`["10.0.0.0/24", "10.0.1.0/24"]`

Trap: `uuid()` and `timestamp()` give a new value on every plan, so a resource argument built from them changes on every run. Use them only where a changing value is the point, and reach for the `random` provider when you want a value generated once and kept.

References

[1] kreuzwerker, "Terraform provider for docker: documentation." 2026. Available: https://github.com/kreuzwerker/terraform-provider-docker/blob/master/docs/index.md

[2] HashiCorp, "Terraform provider for random: documentation." 2026. Available: https://github.com/hashicorp/terraform-provider-random/blob/main/docs/index.md

[3] HashiCorp, "Terraform provider for local: documentation." 2026. Available: https://github.com/hashicorp/terraform-provider-local/blob/main/docs/index.md

[4] HashiCorp, "Terraform provider for local: local_file." 2026. Available: https://github.com/hashicorp/terraform-provider-local/blob/main/docs/resources/file.md

[5] OpenTofu Project, "The lifecycle meta-argument." 2026. Available: https://opentofu.org/docs/language/meta-arguments/lifecycle/

[6] OpenTofu Project, "Custom condition checks." 2026. Available: https://opentofu.org/docs/language/expressions/custom-conditions/

[7] OpenTofu Project, "Module sources." 2026. Available: https://opentofu.org/docs/language/modules/sources/

[8] HashiCorp, "Terraform provider for kubernetes: documentation." 2026. Available: https://github.com/hashicorp/terraform-provider-kubernetes/blob/main/docs/index.md

[9] HashiCorp, "Terraform provider for kubernetes: kubernetes_deployment_v1." 2026. Available: https://github.com/hashicorp/terraform-provider-kubernetes/blob/main/docs/resources/deployment_v1.md

[10] Oracle, "Terraform provider for oracle cloud infrastructure." 2026. Available: https://docs.oracle.com/en-us/iaas/Content/dev/terraform/home.htm

[11] Cloudflare, "Terraform provider for cloudflare." 2026. Available: https://github.com/cloudflare/terraform-provider-cloudflare/blob/master/docs/index.md

[12] C. Marchesi, "Terraform provider for ACME: documentation." 2026. Available: https://github.com/vancluever/terraform-provider-acme/blob/main/docs/index.md

[13] HashiCorp, "Terraform provider for TLS: documentation." 2026. Available: https://github.com/hashicorp/terraform-provider-tls/blob/main/docs/index.md

[14] OpenTofu Project, "OpenTofu documentation." 2026. Available: https://opentofu.org/docs/

[15] HashiCorp, "Terraform documentation." 2026. Available: https://developer.hashicorp.com/terraform/docs

[16] OpenTofu Project, "Module composition." 2026. Available: https://opentofu.org/docs/language/modules/develop/composition/

[17] HashiCorp, "Terraform style guide." 2026. Available: https://developer.hashicorp.com/terraform/language/style

[18] OpenTofu Project, "Standard module structure." 2026. Available: https://opentofu.org/docs/language/modules/develop/structure/

[19] OpenTofu Project, "Style conventions." 2026. Available: https://opentofu.org/docs/language/syntax/style/

[20] OpenTofu Project, "Creating modules." 2026. Available: https://opentofu.org/docs/language/modules/develop/

[21] OpenTofu Project, "Input variables." 2026. Available: https://opentofu.org/docs/language/values/variables/

[22] OpenTofu Project, "The depends_on meta-argument." 2026. Available: https://opentofu.org/docs/language/meta-arguments/depends_on/

[23] OpenTofu Project, "Version constraints." 2026. Available: https://opentofu.org/docs/language/expressions/version-constraints/

[24] OpenTofu Project, "Provider requirements." 2026. Available: https://opentofu.org/docs/language/providers/requirements/

[25] OpenTofu Project, "Dependency lock file." 2026. Available: https://opentofu.org/docs/language/files/dependency-lock/

[26] OpenTofu Project, "Command: Providers lock." 2026. Available: https://opentofu.org/docs/cli/commands/providers/lock/

[27] OpenTofu Project, "Command: init." 2026. Available: https://opentofu.org/docs/cli/commands/init/

[28] OpenTofu Project, "Sensitive data in state." 2026. Available: https://opentofu.org/docs/language/state/sensitive-data/

[29] OpenTofu Project, "Remote state." 2026. Available: https://opentofu.org/docs/language/state/remote/

[30] OpenTofu Project, "State locking." 2026. Available: https://opentofu.org/docs/language/state/locking/

[31] OpenTofu Project, "Backend configuration." 2026. Available: https://opentofu.org/docs/language/settings/backends/configuration/

[32] OpenTofu Project, "Command: state." 2026. Available: https://opentofu.org/docs/cli/commands/state/

[33] OpenTofu Project, "Refactoring." 2026. Available: https://opentofu.org/docs/language/modules/develop/refactoring/

[34] OpenTofu Project, "Import." 2026. Available: https://opentofu.org/docs/language/import/

[35] OpenTofu Project, "Command: State pull." 2026. Available: https://opentofu.org/docs/cli/commands/state/pull/

[36] OpenTofu Project, "Workspaces." 2026. Available: https://opentofu.org/docs/language/state/workspaces/

[37] OpenTofu Project, "Command: plan." 2026. Available: https://opentofu.org/docs/cli/commands/plan/

[38] OpenTofu Project, "Command: apply." 2026. Available: https://opentofu.org/docs/cli/commands/apply/

[39] OpenTofu Project, "The core OpenTofu workflow." 2026. Available: https://opentofu.org/docs/intro/core-workflow/

[40] HashiCorp, "Running Terraform in automation." 2026. Available: https://developer.hashicorp.com/terraform/tutorials/automation/automate-terraform

[41] OpenTofu Project, "Command: destroy." 2026. Available: https://opentofu.org/docs/cli/commands/destroy/

[42] OpenTofu Project, "Environment variables." 2026. Available: https://opentofu.org/docs/cli/config/environment-variables/

[43] OpenTofu Project, "Provider configuration." 2026. Available: https://opentofu.org/docs/language/providers/configuration/

[44] OpenTofu Project, "Sensitive function." 2026. Available: https://opentofu.org/docs/language/functions/sensitive/

[45] HashiCorp, "Terraform provider for random: random_password." 2026. Available: https://github.com/hashicorp/terraform-provider-random/blob/main/docs/resources/password.md

[46] The Kubernetes Authors, "Images." 2026. Available: https://kubernetes.io/docs/concepts/containers/images/

[47] The Kubernetes Authors, "Configuration best practices." 2026. Available: https://kubernetes.io/docs/concepts/configuration/overview/

[48] OpenTofu Project, "Provisioners." 2026. Available: https://opentofu.org/docs/language/resources/provisioners/syntax/

[49] HashiCorp, "The terraform_data managed resource type." 2026. Available: https://developer.hashicorp.com/terraform/language/resources/terraform-data

[50] OpenTofu Project, "Checks with assertions." 2026. Available: https://opentofu.org/docs/language/checks/

[51] OpenTofu Project, "Command: fmt." 2026. Available: https://opentofu.org/docs/cli/commands/fmt/

[52] OpenTofu Project, "Command: validate." 2026. Available: https://opentofu.org/docs/cli/commands/validate/

[53] A. Babenko, "Pre-commit-terraform." 2026. Available: https://github.com/antonbabenko/pre-commit-terraform

[54] TFLint Contributors, "TFLint: A pluggable Terraform linter." 2026. Available: https://github.com/terraform-linters/tflint

[55] Aqua Security, "Trivy." 2026. Available: https://trivy.dev/

[56] Prisma Cloud, "Checkov." 2026. Available: https://www.checkov.io/

[57] terraform-docs Contributors, "Terraform-docs." 2026. Available: https://terraform-docs.io/

[58] tofuutils, "Tenv: OpenTofu and Terraform version manager." 2026. Available: https://github.com/tofuutils/tenv

[59] OpenTofu Project, "OpenTofu language server." 2026. Available: https://github.com/opentofu/tofu-ls

[60] OpenTofu Project, "Command: console." 2026. Available: https://opentofu.org/docs/cli/commands/console/

[61] OpenTofu Project, "Command: show." 2026. Available: https://opentofu.org/docs/cli/commands/show/

[62] OpenTofu Project, "Command: graph." 2026. Available: https://opentofu.org/docs/cli/commands/graph/

[63] OpenTofu Project, "Command: refresh." 2026. Available: https://opentofu.org/docs/cli/commands/refresh/

[64] OpenTofu Project, "Debugging OpenTofu." 2026. Available: https://opentofu.org/docs/internals/debugging/

[65] OpenTofu Project, "Command: test." 2026. Available: https://opentofu.org/docs/cli/commands/test/

[66] OpenTofu Project, "Command: Providers schema." 2026. Available: https://opentofu.org/docs/cli/commands/providers/schema/

[67] OpenTofu Project, "Migrating to OpenTofu from Terraform." 2026. Available: https://opentofu.org/docs/intro/migration/

[68] OpenTofu Project, "State and plan encryption." 2026. Available: https://opentofu.org/docs/language/state/encryption/

[69] HashiCorp, "HCP Terraform documentation." 2026. Available: https://developer.hashicorp.com/terraform/cloud-docs

[70] OpenTofu Project, "What's new in OpenTofu 1.8." 2026. Available: https://opentofu.org/docs/v1.8/intro/whats-new/

[71] HashiCorp, "Terraform stacks." 2026. Available: https://developer.hashicorp.com/terraform/language/stacks

[72] OpenTofu Project, "What's new in OpenTofu 1.9." 2026. Available: https://opentofu.org/docs/v1.9/intro/whats-new/

[73] HashiCorp, "Ephemeral resources." 2026. Available: https://developer.hashicorp.com/terraform/language/resources/ephemeral

[74] HashiCorp, "Write-only arguments." 2026. Available: https://developer.hashicorp.com/terraform/language/resources/ephemeral/write-only

[75] HashiCorp, "Command: query." 2026. Available: https://developer.hashicorp.com/terraform/cli/commands/query

[76] OpenTofu Project, "What's new in OpenTofu 1.10." 2026. Available: https://opentofu.org/docs/v1.10/intro/whats-new/

[77] OpenTofu Project, "OCI registries." 2026. Available: https://opentofu.org/docs/cli/oci_registries/

[78] OpenTofu Project, "What's new in OpenTofu 1.7." 2026. Available: https://opentofu.org/docs/v1.7/intro/whats-new/

[79] OpenTofu Project, "OpenTofu settings." 2026. Available: https://opentofu.org/docs/language/settings/

[80] OpenTofu Project, "Command: Force-unlock." 2026. Available: https://opentofu.org/docs/cli/commands/force-unlock/

[81] OpenTofu Project, "Resource blocks." 2026. Available: https://opentofu.org/docs/language/resources/syntax/

[82] OpenTofu Project, "References to named values." 2026. Available: https://opentofu.org/docs/language/expressions/references/

[83] OpenTofu Project, "Types and values." 2026. Available: https://opentofu.org/docs/language/expressions/types/

[84] OpenTofu Project, "Tostring function." 2026. Available: https://opentofu.org/docs/language/functions/tostring/

[85] OpenTofu Project, "The for_each meta-argument." 2026. Available: https://opentofu.org/docs/language/meta-arguments/for_each/

[86] OpenTofu Project, "Resource graph." 2026. Available: https://opentofu.org/docs/internals/graph/

[87] OpenTofu Project, "Command: import." 2026. Available: https://opentofu.org/docs/cli/import/

[88] Docker Inc., "Networking overview." 2026. Available: https://docs.docker.com/engine/network/

[89] Docker Inc., "Docker container run." 2026. Available: https://docs.docker.com/reference/cli/docker/container/run/

[90] Docker Inc., "Linux post-installation steps for docker engine." 2026. Available: https://docs.docker.com/engine/install/linux-postinstall/

[91] kreuzwerker, "Terraform provider for docker: docker_container." 2026. Available: https://github.com/kreuzwerker/terraform-provider-docker/blob/master/docs/resources/container.md

[92] The Kubernetes Authors, "Debug pods." 2026. Available: https://kubernetes.io/docs/tasks/debug/debug-application/debug-pods/

[93] The Kubernetes Authors, "Kind quick start." 2026. Available: https://kind.sigs.k8s.io/docs/user/quick-start/

[94] The Kubernetes Authors, "Organizing cluster access using kubeconfig files." 2026. Available: https://kubernetes.io/docs/concepts/configuration/organize-cluster-access-kubeconfig/

[95] Oracle, "Always free resources." 2026. Available: https://docs.oracle.com/en-us/iaas/Content/FreeTier/freetier_topic-Always_Free_Resources.htm

[96] Oracle, "Oracle cloud free tier." 2026. Available: https://www.oracle.com/cloud/free/

[97] Oracle, "API errors." 2026. Available: https://docs.oracle.com/en-us/iaas/Content/API/References/apierrors.htm

[98] Cloudflare, "Create API token." 2026. Available: https://developers.cloudflare.com/fundamentals/api/get-started/create-token/

[99] Internet Security Research Group, "Rate limits." 2026. Available: https://letsencrypt.org/docs/rate-limits/

[100] Internet Security Research Group, "Staging environment." 2026. Available: https://letsencrypt.org/docs/staging-environment/

[101] Internet Security Research Group, "Let's encrypt documentation." 2026. Available: https://letsencrypt.org/docs/

[102] C. Marchesi, "Terraform provider for ACME: acme_certificate." 2026. Available: https://github.com/vancluever/terraform-provider-acme/blob/main/docs/resources/certificate.md

[103] Internet Security Research Group, "Challenge types." 2026. Available: https://letsencrypt.org/docs/challenge-types/

[104] Oracle, "Connecting to your linux instance using SSH." 2026. Available: https://docs.oracle.com/en-us/iaas/Content/Compute/Tasks/accessinginstance.htm

[105] Canonical, "Cloud-init documentation." 2026. Available: https://cloudinit.readthedocs.io/en/latest/

[106] OpenTofu Project, "Built-in functions." 2026. Available: https://opentofu.org/docs/language/functions/

[107] OpenTofu Project, "Built-in provider." 2026. Available: https://opentofu.org/docs/language/providers/builtin/

[108] OpenTofu Project, "The terraform_remote_state data source." 2026. Available: https://opentofu.org/docs/language/state/remote-state-data/

[109] OpenTofu Project, "Data sources." 2026. Available: https://opentofu.org/docs/language/data-sources/

[110] OpenTofu Project, "Output values." 2026. Available: https://opentofu.org/docs/language/values/outputs/

[111] OpenTofu Project, "Local values." 2026. Available: https://opentofu.org/docs/language/values/locals/

[112] OpenTofu Project, "Module blocks." 2026. Available: https://opentofu.org/docs/language/modules/syntax/

[113] OpenTofu Project, "The count meta-argument." 2026. Available: https://opentofu.org/docs/language/meta-arguments/count/

[114] OpenTofu Project, "The resource provider meta-argument." 2026. Available: https://opentofu.org/docs/language/meta-arguments/resource-provider/

[115] OpenTofu Project, "Arithmetic and logical operators." 2026. Available: https://opentofu.org/docs/language/expressions/operators/

[116] OpenTofu Project, "Conditional expressions." 2026. Available: https://opentofu.org/docs/language/expressions/conditionals/

[117] OpenTofu Project, "For expressions." 2026. Available: https://opentofu.org/docs/language/expressions/for/

[118] OpenTofu Project, "Splat expressions." 2026. Available: https://opentofu.org/docs/language/expressions/splat/

[119] OpenTofu Project, "Strings and templates." 2026. Available: https://opentofu.org/docs/language/expressions/strings/

[120] OpenTofu Project, "Function calls." 2026. Available: https://opentofu.org/docs/language/expressions/function-calls/

Index