Gorgo Tasting OpenTofu and Terraform

September 08, 2026

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 [1]. 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 [2], [3]. 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 [4] 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 [5]. `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 [6]: 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.

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

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

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

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

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

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