Gorgo Tasting OpenTofu and Terraform

September 08, 2026

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.

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