Gorgo Tasting OpenTofu and Terraform

September 08, 2026

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.