Gorgo Tasting OpenTofu and Terraform

September 08, 2026

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 [1]. 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 [2]. 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.

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

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