Avoiding HCL at All Costs

How To Write tf.json Files

Denis Shatokhin

About Me

- he/him
- devops at Relex Solutions
- reside and work in Helsinki for almost 3 years
- passionate about IaC
- obsessed with niche internet aesthetics

Disclaimers

- this was never been used by me in production
- this presentation is free from any form of generated media (images, ~~code~~, text etc.)
  - except one place where I on purpose tried to show a *complex* terraform code example
- for examples I've been using Terraform Provider for UpCloud
  - maybe it will help me to get some bonuses to my Upcloud account

What Is HCL

HCL stands for **HashiCorp Configuration Language**, this is what we're describing Terraform configuration in.

> **HCL** attempts to strike a compromise between generic serialization formats such as **JSON** and configuration formats built around full programming languages such as Ruby. **HCL** syntax is designed to be easily read and written by humans, and allows declarative logic to permit its use in more complex applications.
>
> **HCL** defines a common configuration model that can be parsed from either its native syntax or *from a well-defined equivalent **JSON** structure*.
>
> — README.md, github.com/hashicorp/hcl

Reusability In Terraform: Dynamic Blocks

> A `dynamic` block acts much like a `for` expression, but produces nested blocks instead of a complex typed value. It iterates over a given complex value, and generates a nested block for each element of that complex value.
>
> — Terraform Documentation, Dynamic Blocks

```
# server.tf
resource "upcloud_server" "example" {
  hostname = "example"
  zone     = "fi-hel1"

  dynamic "network_interface" {
    for_each = var.network_interfaces
    content {
      network_interface {
        type = network_interface.value.type
      }
    }
  }
}
```

```
# variables.tf
variable "network_interfaces" {
  default = {
    primary = {
      type = "public"
    },
    secondary = {
      type = "private"
    }
  }
}
```

The Problem With Dynamic Blocks

Sounds great in theory, but can be really complex in practice. The documentation clearly stated it should be used with caution.

> Overuse of `dynamic` blocks can make configuration hard to read and maintain, so we recommend using them only when you need to hide details in order to build a clean user interface for a re-usable module. *Always write nested blocks out literally where possible*.
>
> — Terraform Documentation, Best Practices for dynamic Blocks

The Problem With Dynamic Blocks

To demonstrate the problem with dynamic blocks better, I'll show a real-world example. Let's define a simple YAML file with the list of our servers first.

The end-user will get this to edit:

```yaml
# servers.yaml
servers:
  - name: server-1
    hostname: server-1
    zone: fi-hel1
    plan: 1xCPU-1GB
    cpu: 2
```

The Problem With Dynamic Blocks

Now the HCL part, where we use `dynamic` blocks and `try()` function for almost every attribute.

We'll get this to maintain:

```
# Load server configuration from YAML file
locals {
  servers_config = yamldecode(file("${path.module}/servers.yaml"))
}

# Create servers based on YAML configuration
resource "upcloud_server" "this" {
  for_each = { for server in local.servers_config.servers : server.name => server }

  # Required attributes
  hostname = each.value.hostname
  zone     = each.value.zone

  # Semi-optional attributes
  plan         = try(each.value.plan, "1xCPU-1GB")
  cpu          = try(each.value.cpu, null)
```

```
  mem          = try(each.value.mem, null)
  boot_order   = try(each.value.boot_order, "disk")
  firewall     = try(each.value.firewall, null)
  host         = try(each.value.host, null)
  hot_resize   = try(each.value.hot_resize, null)
  labels       = try(each.value.labels, {})
  metadata     = try(each.value.metadata, true)
  nic_model    = try(each.value.nic_model, null)
  server_group = try(each.value.server_group, null)
  tags         = try(each.value.tags, [])
  timezone     = try(each.value.timezone, null)
  title        = try(each.value.title, null)
  user_data    = try(each.value.user_data, null)
  video_model  = try(each.value.video_model, null)

  # Dynamic login block
  dynamic "login" {
    for_each = try([each.value.login], [])
    content {
      user              = try(login.value.user, null)
      keys              = try(login.value.keys, null)
      create_password   = try(login.value.create_password, null)
      password_delivery = try(login.value.password_delivery, null)
    }
  }
```

```
# Dynamic network_interface blocks
dynamic "network_interface" {
  for_each = try(each.value.network_interfaces, [
    {
      type = "public"
    }
  ])
  content {
    type                = network_interface.value.type
    index               = try(network_interface.value.index, null)
    ip_address          = try(network_interface.value.ip_address, null)
    ip_address_family   = try(network_interface.value.ip_address_family, null)
    network             = try(network_interface.value.network, null)
    source_ip_filtering = try(network_interface.value.source_ip_filtering, null)
    bootable            = try(network_interface.value.bootable, null)

    # Dynamic additional_ip_address blocks
    dynamic "additional_ip_address" {
      for_each = try(network_interface.value.additional_ip_addresses, [])
      content {
        ip_address        = try(additional_ip_address.value.ip_address, null)
        ip_address_family = try(additional_ip_address.value.ip_address_family, null)
      }
    }
```

```
  }
}

# Dynamic simple_backup block
dynamic "simple_backup" {
  for_each = try([each.value.simple_backup], [])
  content {
    plan = try(simple_backup.value.plan, null)
    time = try(simple_backup.value.time, null)
  }
}

# Dynamic storage_devices blocks
dynamic "storage_devices" {
  for_each = try(each.value.storage_devices, [])
  content {
    storage          = storage_devices.value.storage
    type             = try(storage_devices.value.type, null)
    address          = try(storage_devices.value.address, null)
    address_position = try(storage_devices.value.address_position, null)
  }
}

# Dynamic template block
dynamic "template" {
```

```
    for_each = try([each.value.template], [
      {
        storage = "Ubuntu Server 24.04 LTS (Noble Numbat)"
        size    = 25
      }
    ])
    content {
      storage                  = try(template.value.storage, null)
      size                     = try(template.value.size, null)
      tier                     = try(template.value.tier, null)
      title                    = try(template.value.title, null)
      address                  = try(template.value.address, null)
      address_position         = try(template.value.address_position, null)
      encrypt                  = try(template.value.encrypt, null)
      filesystem_autoresize    = try(template.value.filesystem_autoresize, null)
      delete_autoresize_backup = try(template.value.delete_autoresize_backup, null)

      # Dynamic backup_rule blocks within template
      dynamic "backup_rule" {
        for_each = try(template.value.backup_rules, [])
        content {
          interval  = backup_rule.value.interval
          retention = backup_rule.value.retention
          time      = backup_rule.value.time
        }
```

```
      }
    }
  }
}
```

The changes in provider can cause issues with our codebase and in general it’s cumbersome.

What Is tf.json

In short, tf.json is a JSON file defining terraform configuration.

> The JSON syntax is defined in terms of the native syntax.
>
> *Everything that can be expressed in native syntax can also be expressed in JSON syntax*,
>
> but some constructs are more complex to represent in JSON due to limitations of the JSON grammar.
>
> — Terraform Documentation, JSON Configuration Syntax

tf.json and HCL comparison

Terraform variables

JSON

```
{
  "variable": {
    "hostname": {
      "type": "string",
      "default": "hug"
    },
    "zone": {
      "type": "string",
      "default": "fi-hel1"
    }
  }
}
```

HCL

```
variable "hostname" {
  type    = string
  default = "hug"
}

variable "zone" {
  type    = string
  default = "fi-hel1"
}
```

tf.json and HCL comparison

Terraform resources

JSON

```
{
  "resource": {
    "upcloud_server": {
      "example": {
        "hostname": "${var.hostname}",
        "zone": "${var.zone}"
      }
    }
  },
  "data": {
    "upcloud_networks": {
      "upcloud": {}
    }
  }
}
```

HCL

```
resource "upcloud_server" "example" {
  hostname = var.hostname
  zone     = var.zone
}

data "upcloud_networks" "upcloud" {}
```

tf.json and HCL comparison

Terraform blocks

JSON

```
{
  "resource": {
    "upcloud_server": {
      "example": {
        "hostname": "${var.hostname}",
        "zone": "${var.zone}",
        "network_interface": [
          { "type": "public" },
          { "type": "private", "network": "<network_id>" }
        ]
      }
    }
  }
}
```

HCL

```
resource "upcloud_server" "example" {
  hostname = var.hostname
  zone     = var.zone

  network_interface {
    type = "public"
  }

  network_interface {
    type = "private"
    network = "<network_id>"
  }
}
```

Ways To Generate tf.json: YAML

Technically YAML is a superset of JSON so for generating the `.tf.json` file we will use `mikefarah/yq` - CLI tool to query and write YAML files

```
$ yq --output-format json main.tf.yaml > main.tf.json
$ terraform init
$ terraform plan
$ terraform plan -var hostname=example
```

This approach also gives us a way to modify the generated `main.tf.json` file, for example, to change the version of the provider

```
$ yq --output-format json
'.terraform.required_providers.upcloud.version = "~> 5.28.0"'
main.tf.yaml > main.tf.json
```

```yaml
# main.tf.yaml
terraform:
  required_providers:
    upcloud:
      source: UpCloudLtd/upcloud
      version: ~> 5.26.0

provider:
  upcloud: {}

variable:
  hostname:
    type: string
    default: hug

resource:
  upcloud_server:
    example:
      hostname: ${var.hostname}
      zone: fi-hel1
```

Ways To Generate tf.json: PKL

Build by Apple and open-sourced in 2024.

> Pkl — pronounced *Pickle* — is a configuration-as-code language with rich validation and tooling. It can be used as a command line tool, software library, or build plugin. Pkl scales from small to large, simple to complex, ad-hoc to recurring configuration tasks.
>
> — PKL Website, Introduction

- no Apple ID required
- turing complete
- written in `java/kotlin`
- LSP support
- community growing fast

Ways To Generate tf.json: Simple PKL

HCL

```
variable "hostname" {
  type    = string
  default = "hug"
}

resource "upcloud_server" "example" {
  hostname = var.hostname
  zone     = "fi-hel1"
}
```

PKL

```
variable {
  hostname {
    type = "string"
    ["default"] = "hug"
  }
}

resource {
  upcloud_server {
    example {
      hostname = "${var.hostname}"
      zone = "fi-hel1"
    }
  }
}
```

Ways To Generate tf.json: Simple PKL

Here we will use `apple/pkl` - CLI tool for evaluating `.pkl` files

```
$ pkl eval --format json main.tf.pkl --output-path main.tf.json
$ terraform init
$ terraform plan -var hostname=example
```

```
/// main.tf.pkl
terraform {
  required_providers {
    upcloud {
      source = "UpCloudLtd/upcloud"
      version = "~> 5.28.0"
    }
  }
}

provider { upcloud {} }
variable { hostname {} }

resource {
  upcloud_server {
    example {
      hostname = "${var.hostname}"
      zone = "fi-hel1"
    }
  }
}
```

Ways To Generate tf.json: Typed PKL

```
class Variable {
  type: String?
  default: Any?
  description: String?
}
function variableRef(name: String): String = "${var.\(name)}"

variable {
  hostname = new Variable {
    type = "string"
    default = "hug"
  }
}
resource {
  upcloud_server {
    example {
      hostname = variableRef("hostname")
      zone = "fi-hel1"
    }
  }
}
```

Now we can utilise types in PKL and define functions

```
{
  "variable": {
    "hostname": {
      "type": "string",
      "default": "hug"
    }
  },
  "resource": {
    "upcloud_server": {
      "example": {
        "hostname": "${var.hostname}",
        "zone": "fi-hel1"
      }
    }
  }
}
```

Integration With Terraform Cloud

Due to additional generating step the **Version Control Workflow** in TFC not feasible.

The **CLI-Driven Workflow** have to used instead, which gives us the same result with a bit more steps.

Let’s see it in action for the examples we’ve just discussed.

Other Configuration Languages

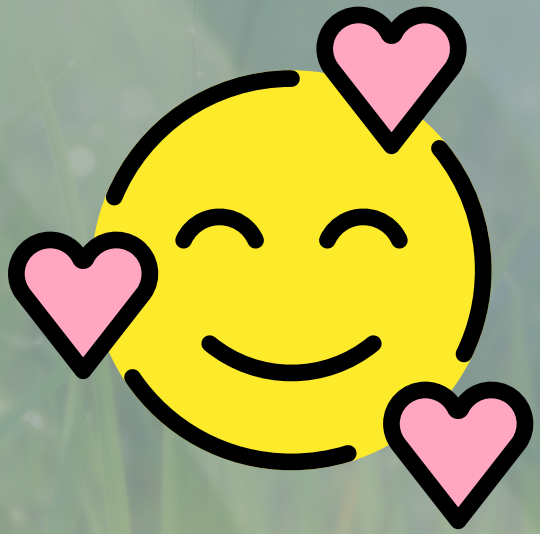
- Jsonnet
- Dhall
- CUE
- Nickel
- KCL

Links

- pkl-lang.org
- pkl-playground.vercel.app
- github.com/dshatokhin/terraform-json

Thank You

For being such an amazing audience



Questions

If you have any left