HashiCorp Terraform Coding Guidelines

Code Review

River Point Technology

March 2021

RPT River Point Technology

1 Terraform Coding Best Practices

1.1 Purpose

This document serves as a guidepost for effective development with Terraform across multiple team members and workstreams. Our goal is to provide a unifed and consistent approach to managing code by outlining some rules of engagement. These guidelines should be referenced during code reviews in order to align new development with existing work and to update old code. As a disclaimer, the guidelines found here are not law but rather best practices that we have encountered in our numerous infrastructure-as-code experiences. In cases where new needs require deviation from this guide, we should adjust the guide.

1.2 Intended Audience

These standards are targeted towards developers and administrators who will be writing Terraform code on a regular basis.

1.3 Terraform Modules

The following guidelines apply to reusable Terraform modules and root configs.

1.3.1 Module Structure

1. Terraform modules must follow the standard module structure.
2. Every module must start with a `main.tf` file, where resources are located by default.
    - Among other resources, all locals should be defined here in a single locals block.
3. All modules must have a `README.md` (with basic documentation in Markdown format).
4. Examples should be located in `examples/`, each with its own subdirectory and a `README.md` file
5. Logical groupings of resources can be grouped into their own files and given descriptive names, like `network.tf`, `instances.tf`, or `loadbalancer.tf`.
    - Avoid giving every resource its own file. Resources should be grouped by shared purpose. For example, "google_dns_managed_zone" and"google_dns_record_set" would likely be combined in `dns.tf`.
6. Only Terraform (`*.tf`) and repo metadata files (like `README.md`, `CHANGELOG.md` or `kitchen.yml`) should exist at the root directory of a module.

7. Additional documentation should be stored in a `docs/` subdirectory.

1.3.2 Naming Convention

All configuration objects should be named using underscores to delimit multiple words. This practice ensures consistency with the naming convention for resource types, data source types, and other predefined values. Note that this convention does not apply to name arguments.

```
# Good
resource "google_compute_instance" "web_server" {
  name = “web-server”
# ... }

# Bad
resource “google_compute_instance” “web-server” {
  name = “web-server”
#... }
```

1.3.3 Variables

1. All variables shall be declared in `variables.tf`.
2. Variables shall have descriptive names relevant to their usage or purpose.
    - Inputs, local variables, and outputs representing numeric values such as disk sizes or RAM size SHOULD be named with units (like ram_size_gb). APIs typically do not have standard units, so naming variables with units makes the expected input unit clear for configuration maintainers.
    - For units of storage, the unit prefix (kilo, mega, giga) SHOULD be binary (powers of 1024). For all other units of measurement, the unit prefix SHOULD be decimal (powers of 1000).
    - Boolean variables SHOULD be named with positive values (like enable_external_access) to simplify conditional logic.
3. Variables must have descriptions. These are automatically included in any published modules’ auto-generated documentation through terraform-docs. Descriptions add additional context for new developers that descriptive names cannot.
4. Variables should have defined types.
5. Variables with non-environment-specific values (like disk size) should be given default values.
6. Variables for environment-specific values (like project_id) should not be given defaults. This forces the calling module to provide meaningful values.

7. Variables should only have empty defaults (like empty strings or lists) where leaving the variable empty is a valid preference which will not be rejected by the underlying API(s).
8. Be thoughtful in your use of static literals (hardcoded strings, etc.) and parameterize anything which must vary per instance or environment. Local values can be used in cases where a literal is reused in multiple places without exposing it as a variable.
9. When deciding whether to expose a variable, ensure that you have a concrete use case for changing that variable. Don't expose variables on the off chance that it's needed.
    - Adding a variable with a default value is backwards compatible, and thus "cheap."
    - Removing a variable is backwards incompatible, and thus "expensive."
10. Boolean variables should be explicitly declared as strings until Terraform adds further support for boolean input variables.

1.3.4 Outputs

1. All outputs shall be organized into `outputs.tf`.
2. Outputs should have meaningful descriptions.
3. Output descriptions should be documented in the README. Descriptions should also be auto-generated on commit with terraform-docs.
4. Make an effort to output all the useful values root modules would want to reference or share with modules. Particularly for open source or heavily used modules, expose all outputs that have potential for consumption.

1.3.5 Data Sources

1. Data sources are located adjacent to the resources which reference them.

- If you are fetching an image to be used in launching an instance, you can place it alongside the instance instead of collecting data resources in their own file.

2. If the number of data sources grows considerably, it's a reasonable practice to move these to a dedicated `data.tf` file.
3. Data sources should use variable or resource interpolation, where appropriate, to fetch data relative to your current environment.

1.3.6 Scripts (Called by Terraform)

1. Bespoke scripts can be called by Terraform through provisioners, including the local-exec provisioner.

2. Custom scripts should be avoided, if possible, and constrained to instances where native Terraform resources do not support the desired behavior. Any custom scripts used must have a clearly documented reasoning and ideally a deprecation plan. Additionally, custom scripts should not replace configuration management tools in cases where they are more appropriate.
3. Bespoke scripts called by Terraform must be organized into `scripts/`.
4. Use scripts only when absolutely necessary, as the state of resources created through scripts is not accounted for or managed by Terraform. You'll likely want to add a policy of `ignore_changes = [*]` on such resources.

1.3.7 Helper Scripts (not called by Terraform)

1. Helper scripts should be organized in a `./helpers` directory.
2. Helper scripts should be documented in the README with an explanation and example invocations.
3. Helper scripts accepting arguments should provide argument-checking and `--help` output.

1.3.8 Static Files

1. Static files which are referenced by Terraform (like Startup scripts loaded onto instances) but not executed must be organized into `files/`.
2. Lengthy heredocs should be externalized from their HCL and into external files. These should be referenced with the `file()` function.

1.3.9 Templates

1. Files which are injected with the Terraform `template_file` resource should be given the file extension `.tpl`.
2. Templates must be placed in `templates/`.

1.3.10 Resources

1. Resources that are the only one of their type (i.e., a single load balancer for an entire module) should be named 'main' to simplify references to that resource. It takes extra mental work to remember: `some_google_resource.my_unique_name.id` vs. `some_google_resource.main.id`
2. Resources that share the same type as others in the same module should be given meaningful names to differentiate them.

3. Resources must be named in snake-case (like db_instance).
4. Resource names should be singular.
5. Resource names shouldn't repeat the resource type within the name. For example: Do this: `resource "google_compute_global_address "main" {...}` Not this: `resource "google_compute_global_address" "main_global_address" { ...}`
6. Ensure that deletion protection is enabled for stateful resources like databases. For example:

```
resource "google_sql_database_instance" "main" { name = "master-
instance"
settings {
tier = "D0" }
  lifecycle {
    prevent_destroy = true
}
}
```

1.3.11 Formatting

1. All Terraform files must conform to the standards of `terraform fmt`.

1.3.12 Expressions

1. Limit the complexity of any individual interpolated expressions. If many functions are needed in a single expression, consider splitting it out into multiple expressions using locals.
2. Never have more than one ternary operation in a single line. Instead, use multiple local values to build up the logic.
3. Be sparing when using user-specified variables to set the `count` variable for resources. If a resource attribute is provided for such a variable (like `project_id`) and that resource does not yet exist, Terraform will not be able to generate a plan and will report the error "value of count cannot be computed."
4. Use count to instantiate a resource conditionally. For example:

```
variable "readers" {
    description = "..."
    type = "list"
    default = []
}
```

```
resource "foo" "bar" {
    // Do not create this resource if the list of readers is empty.
    count = "${length(var.readers) == 0 ? 0 : 1}"
...
}
```

1.4 Common Modules

Modules that are meant for reuse should follow the following standards, as well as the normal Terraform guidelines.

1.4.1 Structure

1. All common modules should have an OWNERSfile (or CODEOWNERS on GitHub) documenting who is responsible for the module.
2. Common modules should follow SemVer v2.0.0 when new versions are tagged/released.
3. Modules must not declare providers or backends. Leave that to the root modules.
    - Working examples should codify if a specific provider version is needed for a given module.

1.4.2 Variables

It's a good practice to allow flexibility in the labelling of resources through the module's interface. Consider providing a `labels` variable with a default value of an empty map to apply throughout labelable resources:

```
variable "labels" {
description = "A map of labels to apply to contained
resources."
default = {} type = "map"
}
```

1.4.3 Outputs

Outputs are required for common modules that define resources.

- Variables and outputs are used to infer dependencies between modules and resources. Without any outputs, users cannot properly order your module in relation to their Terraform configurations.

- Every resource defined in a common module should have at least one output which references that resource.

1.4.4 Inline modules

1. Inline modules may be used to organize complex Terraform modules into smaller units, or de-duplicate common resources.
2. Inline modules shall be placed in `modules/$modulename`.
3. Inline modules should be treated as private and should not be used by outside modules, unless the common module specifically documents them otherwise.
4. Be aware that Terraform doesn't track refactored resources; if you start out with a number of resources in the top level module and then push them into submodules,Terraform will try to recreate all refactored resources.
5. Outputs defined by internal modules are not automatically exposed; if you want to share outputs from internal modules you'll need to re-output them.

1.5 Root Configs

Root configs, or root modules, are the working directories from which you run the Terraform CLI. They should follow the following standards, as well as the normal Terraform guidelines where applicable. Explicit recommendations for root modules supersede the general guidelines. Resources for different applications and projects should be separated into their own Terraform directories that can be managed independently of each other. A service might represent a particular application or a common service like shared networking. Importantly, all the Terraform code for a particular service should be nested under one directory (including subdirectories).

1.5.1 Directory Structure

There are multiple ways to organize Terraform root configurations, especially when it comes to managing multiple environments. When it comes to managing the Terraform config for a particular service, the recommended structure is to use environment directories.

Directories per environment In this style, each service must split its Terraform config into multiple directories. In this structure, the directory layout must be as follows:

```
-- SERVICE-DIRECTORY/
   -- OWNERS
   -- modules/
      -- service/
         -- main.tf
         -- variables.tf
         -- outputs.tf
         -- README
-- ...other... -- environments/
      -- dev/
         -- backend.tf
         -- main.tf
         -- provider.tf
      -- qa/
         -- backend.tf
         -- main.tf
         -- provider.tf
      -- prod/
         -- backend.tf
         -- main.tf
         -- provider.tf
```

Environment directories Each environment directory within corresponds to a Terraform Workspace and deploys a version of the service to that environment. This config should reference modules to share code across environments, including typically a service module which includes the base shared Terraform config for the service.

This environment directory must contain the following files: * A `backend.tf` file declaring the Terraform backend state location (typically GCS). * A `main.tf` file which instantiates the service module. * A `provider.tf` file which declares provider configuration.

Workspaces per environment Alternatively, a single Terraform directory can be used per service and shared across environments. Each environment would have its own workspace. When using workspaces, all environments share the same modules, and the configuration is driven by a tfvars file and a workspace. Workspaces are helpful in that they limit the amount of code that must be copy-pasted between environment directories, which can help enforce parity between environments while maintaining their own state files. By default a single workspace named “default” exists. It is recommended to create and use a workspace for each environment and use the “default” workspace only when working with resources that may be used across multiple environments like some service accounts.

1.5.2 Outputs

1. Information from a root module which other root modules may depend on must be exported as outputs.
2. Root module outputs can be referenced using remote state.

Publishing outputs with remote states

1. Make sure to re-output nested module outputs that are useful as remote state. Only root module-level outputs can be referenced from other Terraform environments/applications.
2. Information related to a service's endpoints should be exported to remote state to allow use by other dependent apps for configuration.

1.6 Versioning

1.6.1 Terraform

Terraform v0.12 is a significant release that will include some backwards incompatibilities. Pin the Terraform version (example given below) to a known safe version until v0.12 has been released and stabilized.

```
terraform {
required_version = "~> 0.11.10"
}
```

1.6.2 Providers

Pin any providers to a known good version, and make updating the version pin a regular practice.

```
provider "google" { version = "~> 1.19.1"
}
```

1.6.3 Modules

References to shared modules must be constrained to a release tag. Targeting a specific commit hash or branch is dangerous as it gives no context to the version of the underlying module. Updating modules should involve as little guesswork as possible for both authors and reviewers.

1.6.4 Constrain by git reference

References to shared modules may be constrained to any arbitrary git reference (commit, branch, or tag). For reasons outlined above, we only recommend using this to reference tags:

```
module "vpc" {
  source = “git::https://github.com/terraform-google-modules/terraform-
google-network?ref=v0.4.0”
... }
```

1.6.5 Constrain by version

When a git tag is released to the Terraform Module Registry, it creates a numbered version of that module (note that this does not apply to Github repositories, only to modules released to registries). An invocation can be constrained to said version:

```
module "nat_gateway" {
  source = “GoogleCloudPlatform/nat-gateway/google”
  version = “1.2.2”
  ...
}
```