English

Welcome

This document is an attempt to systematically describe best practices using Terraform and provide recommendations for the most frequent problems Terraform users experience.

Terraform is powerful (if not the most powerful out there now) and one of the most used tools which allow management of infrastructure as code. It allows developers to do a lot of things and does not restrict them from doing things in ways that will be hard to support or integrate with.

Some information described in this book may not seem like the best practices. I know this, and to help readers to separate what are established best practices and what is just another opinionated way of doing things, I sometimes use hints to provide some context and icons to specify the level of maturity on each subsection related to best practices.

The book was started in sunny Madrid in 2018, available for free here at https://www.terraform-best-practices.com/.

A few years later it has been updated with more actual best practices available with Terraform 1.0. Eventually, this book should contain most of the indisputable best practices and recommendations for Terraform users.

Translations

Español (Spanish)

Bahasa Indonesia (Indonesian)

Bosanski (Bosnian)

Français (French)

Deutsch (German)

(Hebrew) עברית

Italiano (Italian)

(Kannada)

हिंदी (Hindi)

Português (Brazilian Portuguese)

Polski (Polish)

Română (Romanian)

Türkçe (Turkish)

Українська (Ukrainian)

Contact me if you want to help translate this book into other languages.

Contributions

I always want to get feedback and update this book as the community matures and new ideas are implemented and verified over time.

If you are interested in specific topics, please open an issue, or thumb up an issue you want to be covered. If you feel that **you have content** and you want to contribute, write a draft and submit a pull request (don't worry about writing good text at this point!).

Authors

This book is maintained by Anton Babenko with the help of different contributors and translators.

Sponsors

CLUSTER.DEV	Cluster.dev — the only manager for cloud-native infrastructures.
Coder	Coder.com — create remote development machines for your team, powered by Terraform.

License

Key concepts

The official Terraform documentation describes all aspects of configuration in details. Read it carefully to understand the rest of this section.

This section describes key concepts which are used inside the book.

Resource

Resource is `aws_vpc`, `aws_db_instance`, etc. A resource belongs to a provider, accepts arguments, outputs attributes, and has a lifecycle. A resource can be created, retrieved, updated, and deleted.

Resource module

Resource module is a collection of connected resources which together perform the common action (for e.g., AWS VPC Terraform module creates VPC, subnets, NAT gateway, etc). It depends on provider configuration, which can be defined in it, or in higher-level structures (e.g., in infrastructure module).

Infrastructure module

An infrastructure module is a collection of resource modules, which can be logically not connected, but in the current situation/project/setup serves the same purpose. It defines the configuration for providers, which is passed to the downstream resource modules and to resources. It is normally limited to work in one entity per logical separator (e.g., AWS Region, Google Project).

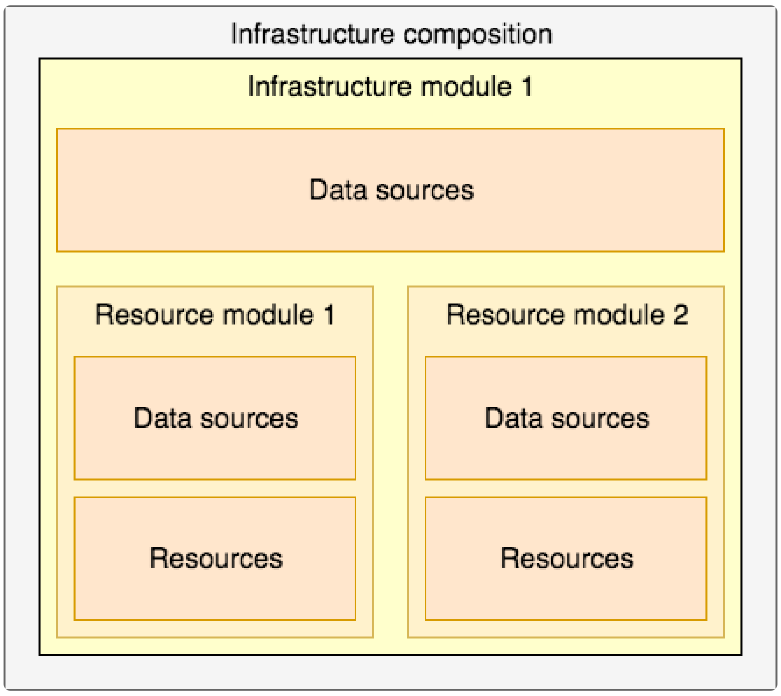
For example, terraform-aws-atlantis module uses resource modules like terraform-aws-vpc and terraform-aws-security-group to manage the infrastructure required for running Atlantis on AWS Fargate.

Another example is terraform-aws-cloudquery module where multiple modules by terraform-aws-modules are being used together to manage the infrastructure as well as using Docker resources to build, push, and deploy Docker images. All in one set.

Composition

Composition is a collection of infrastructure modules, which can span across several logically separated areas (e.g.., AWS Regions, several AWS accounts). Composition is used to describe the complete infrastructure required for the whole organization or project.

A composition consists of infrastructure modules, which consist of resources modules, which implement individual resources.



Simple infrastructure composition

Data source

Data source performs a read-only operation and is dependant on provider configuration, it is used in a resource module and an infrastructure module.

Data source `terraform_remote_state` acts as a glue for higher-level modules and compositions.

The external data source allows an external program to act as a data source, exposing arbitrary data for use elsewhere in the Terraform configuration. Here is an example from the terraform-aws-lambda module where the filename is computed by calling an external Python script.

The http data source makes an HTTP GET request to the given URL and exports information about the response which is often useful to get information from endpoints where a native Terraform provider does not exist.

Remote state

Infrastructure modules and compositions should persist their Terraform state in a remote location where it can be retrieved by others in a controllable way (e.g., specify ACL, versioning, logging).

Provider, provisioner, etc

Providers, provisioners, and a few other terms are described very well in the official documentation and there is no point to repeat it here. To my opinion, they have little to do with writing good Terraform modules.

Why so *difficult*?

While individual resources are like atoms in the infrastructure, resource modules are molecules (consisting of atoms). A module is the smallest versioned and shareable unit. It has an exact list of arguments, implement basic logic for such a unit to do the required function. e.g., terraform-aws-security-group module creates `aws_security_group` and `aws_security_group_rule` resources based on input. This resource module by itself can be used together with other modules to create the infrastructure module.

Access to data across molecules (resource modules and infrastructure modules) is performed using the modules' outputs and data sources.

Access between compositions is often performed using remote state data sources. There are multiple ways to share data between configurations.

When putting concepts described above in pseudo-relations it may look like this:

```
composition-1 {
  infrastructure-module-1 {
    data-source-1 => d1

    resource-module-1 {
      data-source-2 => d2
      resource-1 (d1, d2)
      resource-2 (d2)
    }

    resource-module-2 {
      data-source-3 => d3
      resource-3 (d1, d3)
      resource-4 (d3)
    }
  }
}
```

Code structure

Questions related to Terraform code structure are by far the most frequent in the community. Everyone thought about the best code structure for the project at some point also.

How should I structure my Terraform configurations?

This is one of the questions where lots of solutions exist and it is very hard to give universal advice, so let's start with understanding what are we dealing with.

- What is the complexity of your project?
  - Number of related resources
  - Number of Terraform providers (see note below about "logical providers")
- How often does your infrastructure change?
  - **From** once a month/week/day
  - **To** continuously (every time when there is a new commit)
- Code change initiators? *Do you let the CI server update the repository when a new artifact is built?*
  - Only developers can push to the infrastructure repository
  - Everyone can propose a change to anything by opening a PR (including automated tasks running on the CI server)
- Which deployment platform or deployment service do you use?
  - AWS CodeDeploy, Kubernetes, or OpenShift require a slightly different approach
- How environments are grouped?
  - By environment, region, project

> *Logical providers* work entirely within Terraform's logic and very often don't interact with any other services, so we can think about their complexity as O(1). The most common logical providers include random, local, terraform, null, time.

Getting started with the structuring of Terraform configurations

Putting all code in `main.tf` is a good idea when you are getting started or writing an example code. In all other cases you will be better having several files split logically like this:

- `main.tf` - call modules, locals, and data sources to create all resources
- `variables.tf` - contains declarations of variables used in `main.tf`
- `outputs.tf` - contains outputs from the resources created in `main.tf`
- `versions.tf` - contains version requirements for Terraform and providers

`terraform.tfvars` should not be used anywhere except composition.

How to think about Terraform configuration structure?

Please make sure that you understand key concepts - resource module, infrastructure module, and composition, as they are used in the following examples.

Common recommendations for structuring code

- It is easier and faster to work with a smaller number of resources
  - `terraform plan` and `terraform apply` both make cloud API calls to verify the status of resources
  - If you have your entire infrastructure in a single composition this can take some time
- A blast radius (in case of security breach) is smaller with fewer resources
  - Insulating unrelated resources from each other by placing them in separate compositions reduces the risk if something goes wrong
- Start your project using remote state because:
  - Your laptop is no place for your infrastructure source of truth
  - Managing a `tfstate` file in git is a nightmare
  - Later when infrastructure layers start to grow in multiple directions (number of dependencies or resources) it will be easier to keep things under control
- Practice a consistent structure and naming convention:
  - Like procedural code, Terraform code should be written for people to read first, consistency will help when changes happen six months from now
  - It is possible to move resources in Terraform state file but it may be harder to do if you have inconsistent structure and naming
- Keep resource modules as plain as possible
- Don't hardcode values that can be passed as variables or discovered using data sources
- Use data sources and `terraform_remote_state` specifically as a glue between infrastructure modules within the composition

In this book, example projects are grouped by *complexity* - from small to very-large infrastructures. This separation is not strict, so check other structures also.

Orchestration of infrastructure modules and compositions

Having a small infrastructure means that there is a small number of dependencies and few resources. As the project grows the need to chain the execution of Terraform configurations, connecting different infrastructure modules, and passing values within a composition becomes obvious.

There are at least 5 distinct groups of orchestration solutions that developers use:

1. Terraform only. Very straightforward, developers have to know only Terraform to get the job done.
2. Terragrunt. Pure orchestration tool which can be used to orchestrate the entire infrastructure as well as handle dependencies. Terragrunt operates with infrastructure modules and compositions natively, so it reduces duplication of code.
3. In-house scripts. Often this happens as a starting point towards orchestration and before discovering Terragrunt.
4. Ansible or similar general purpose automation tool. Usually used when Terraform is adopted after Ansible, or when Ansible UI is actively used.
5. Crossplane and other Kubernetes-inspired solutions. Sometimes it makes sense to utilize the Kubernetes ecosystem and employ a reconciliation loop feature to achieve the desired state of your Terraform configurations. View video Crossplane vs Terraform for more information.

With that in mind, this book reviews the first two of these project structures, Terraform only and Terragrunt.

See examples of code structures for Terraform or Terragrunt in the next chapter.

Code structure examples

Terraform code structures

These examples are showing AWS provider but the majority of principles shown in the examples can be applied to other public cloud providers as well as other kinds of providers (DNS, DB, Monitoring, etc)

Type	Description	Readiness
small	Few resources, no external dependencies. Single AWS account. Single region. Single environment.	Yes
medium	Several AWS accounts and environments, off-the-shelf infrastructure modules using Terraform.	Yes
large	Many AWS accounts, many regions, urgent need to reduce copy-paste, custom infrastructure modules, heavy usage of compositions. Using Terraform.	WIP
very-large	Several providers (AWS, GCP, Azure). Multi-cloud deployments. Using Terraform.	No

Terragrunt code structures

Type	Description	Readiness
medium	Several AWS accounts and environments, off-the-shelf infrastructure modules, composition pattern using Terragrunt.	No
large	Many AWS accounts, many regions, urgent need to reduce copy-paste, custom infrastructure modules, heavy usage of compositions. Using Terragrunt.	No
very-large	Several providers (AWS, GCP, Azure). Multi-cloud deployments. Using Terragrunt.	No

Terragrunt

Terraform

Small-size infrastructure with Terraform

Source: https://github.com/antonbabenko/terraform-best-practices/tree/master/examples/small-terraform

This example contains code as an example of structuring Terraform configurations for a small-size infrastructure, where no external dependencies are used.

- Perfect to get started and refactor as you go
- Perfect for small resource modules
- Good for small and linear infrastructure modules (eg, terraform-aws-atlantis)
- Good for a small number of resources (fewer than 20-30)

Single state file for all resources can make the process of working with Terraform slow if the number of resources is growing (consider using an argument `-target` to limit the number of resources)

Medium-size infrastructure with Terraform

Source: https://github.com/antonbabenko/terraform-best-practices/tree/master/examples/medium-terraform

This example contains code as an example of structuring Terraform configurations for a medium-size infrastructure which uses:

- 2 AWS accounts
- 2 separate environments (`prod` and `stage` which share nothing). Each environment lives in a separate AWS account
- Each environment uses a different version of the off-the-shelf infrastructure module (`alb`) sourced from Terraform Registry
- Each environment uses the same version of an internal module `modules/network` since it is sourced from a local directory.

> - Perfect for projects where infrastructure is logically separated (separate AWS accounts)
> - Good when there is no is need to modify resources shared between AWS accounts (one environment = one AWS account = one state file)
> - Good when there is no need in the orchestration of changes between the environments
> - Good when infrastructure resources are different per environment on purpose and can't be generalized (eg, some resources are absent in one environment or in some regions)

> As the project grows, it will be harder to keep these environments up-to-date with each other. Consider using infrastructure modules (off-the-shelf or internal) for repeatable tasks.

Large-size infrastructure with Terraform

Source: https://github.com/antonbabenko/terraform-best-practices/tree/master/examples/large-terraform

This example contains code as an example of structuring Terraform configurations for a large-size infrastructure which uses:

- 2 AWS accounts
- 2 regions
- 2 separate environments (`prod` and `stage` which share nothing). Each environment lives in a separate AWS account and span resources between 2 regions
- Each environment uses a different version of the off-the-shelf infrastructure module (`alb`) sourced from Terraform Registry
- Each environment uses the same version of an internal module `modules/network` since it is sourced from a local directory.

In a large project like described here the benefits of using Terragrunt become very visible. See Code Structures examples with Terragrunt.

- Perfect for projects where infrastructure is logically separated (separate AWS accounts)
- Good when there is no is need to modify resources shared between AWS accounts (one environment = one AWS account = one state file)
- Good when there is no need for the orchestration of changes between the environments
- Good when infrastructure resources are different per environment on purpose and can't be generalized (eg, some resources are absent in one environment or in some regions)

As the project grows, it will be harder to keep these environments up-to-date with each other. Consider using infrastructure modules (off-the-shelf or internal) for repeatable tasks.

Naming conventions

General conventions

> ⓘ There should be no reason to not follow at least these conventions :)

> ⓘ Beware that actual cloud resources often have restrictions in allowed names. Some resources, for example, can't contain dashes, some must be camel-cased. The conventions in this book refer to Terraform names themselves.

1. Use `_` (underscore) instead of `-` (dash) everywhere (in resource names, data source names, variable names, outputs, etc).
2. Prefer to use lowercase letters and numbers (even though UTF-8 is supported).

Resource and data source arguments

1. Do not repeat resource type in resource name (not partially, nor completely):

> ✓ `` `resource "aws_route_table" "public" {}` ``

> ⚠ `` `resource "aws_route_table" "public_route_table" {}` ``

> ⚠ `` `resource "aws_route_table" "public_aws_route_table" {}` ``

1. Resource name should be named `this` if there is no more descriptive and general name available, or if the resource module creates a single resource of this type (eg, in AWS VPC module there is a single resource of type `aws_nat_gateway` and multiple resources of type `aws_route_table`, so `aws_nat_gateway` should be named `this` and `aws_route_table` should have more descriptive names - like `private`, `public`, `database`).
2. Always use singular nouns for names.
3. Use `-` inside arguments values and in places where value will be exposed to a human (eg, inside DNS name of RDS instance).
4. Include argument `count` / `for_each` inside resource or data source block as the first argument at

the top and separate by newline after it.

5. Include argument `tags,` if supported by resource, as the last real argument, following by `depends_on` and `lifecycle`, if necessary. All of these should be separated by a single empty line.
6. When using conditions in an argument `count` / `for_each` prefer boolean values instead of using `length` or other expressions.

Code examples of `resource`

Usage of `count` / `for_each`

✓ main.tf

```
resource "aws_route_table" "public" {
  count = 2

  vpc_id = "vpc-12345678"
  # ... remaining arguments omitted
}

resource "aws_route_table" "private" {
  for_each = toset(["one", "two"])

  vpc_id = "vpc-12345678"
  # ... remaining arguments omitted
}
```

⚠ main.tf

```
resource "aws_route_table" "public" {
  vpc_id = "vpc-12345678"
  count  = 2

  # ... remaining arguments omitted
}
```

Placement of `tags`

✓ main.tf

```
resource "aws_nat_gateway" "this" {
  count = 2

  allocation_id = "..."
  subnet_id     = "..."


  tags = {
    Name = "..."
  }

  depends_on = [aws_internet_gateway.this]

  lifecycle {
    create_before_destroy = true
  }
}
```

⚠ main.tf

```
resource "aws_nat_gateway" "this" {
  count = 2

  tags = "..."

  depends_on = [aws_internet_gateway.this]

  lifecycle {
    create_before_destroy = true
  }

  allocation_id = "..."
  subnet_id     = "..."
}
```

Conditions in `count`

outputs.tf

```
resource "aws_nat_gateway" "that" {    # Best
  count = var.create_public_subnets ? 1 : 0
}


resource "aws_nat_gateway" "this" {    # Good
  count = length(var.public_subnets) > 0 ? 1 : 0
}
```

Variables

1. Don't reinvent the wheel in resource modules: use `name`, `description`, and `default` value for variables as defined in the "Argument Reference" section for the resource you are working with.
2. Support for validation in variables is rather limited (e.g. can't access other variables or do lookups). Plan accordingly because in many cases this feature is useless.
3. Use the plural form in a variable name when type is `list(...)` or `map(...)`.
4. Order keys in a variable block like this: `description` , `type`, `default`, `validation`.
5. Always include `description` on all variables even if you think it is obvious (you will need it in the future).
6. Prefer using simple types (`number`, `string`, `list(...)`, `map(...)`, `any`) over specific type like `object()` unless you need to have strict constraints on each key.
7. Use specific types like `map(map(string))` if all elements of the map have the same type (e.g. `string`) or can be converted to it (e.g. `number` type can be converted to `string`).
8. Use type `any` to disable type validation starting from a certain depth or when multiple types should be supported.
9. Value `{}` is sometimes a map but sometimes an object. Use `tomap(...)` to make a map because there is no way to make an object.

Outputs

Make outputs consistent and understandable outside of its scope (when a user is using a module it should be obvious what type and attribute of the value it returns).

1. The name of output should describe the property it contains and be less free-form than you would normally want.
2. Good structure for the name of output looks like `{name}_{type}_{attribute}` , where:
    1. `{name}` is a resource or data source name without a provider prefix. `{name}` for `aws_subnet` is `subnet`, for `aws_vpc` it is `vpc`.
    2. `{type}` is a type of a resource sources
    3. `{attribute}` is an attribute returned by the output

    4. See examples.

3. If the output is returning a value with interpolation functions and multiple resources, `{name}` and `{type}` there should be as generic as possible (`this` as prefix should be omitted). See example.
4. If the returned value is a list it should have a plural name. See example.
5. Always include `description` for all outputs even if you think it is obvious.
6. Avoid setting `sensitive` argument unless you fully control usage of this output in all places in all modules.
7. Prefer `try()` (available since Terraform 0.13) over `element(concat(...))` (legacy approach for the version before 0.13)

Code examples of `output`

Return at most one ID of security group:

✓ outputs.tf

```
output "security_group_id" {
  description = "The ID of the security group"
  value       = try(aws_security_group.this[0].id, aws_security_group.name_prefix[0]
}
```

When having multiple resources of the same type, `this` should be omitted in the name of output:

⚠ outputs.tf

```
output "this_security_group_id" {
  description = "The ID of the security group"
  value       = element(concat(coalescelist(aws_security_group.this.*.id, aws_securi
}
```

Use plural name if the returning value is a list

✓ outputs.tf

```
output "rds_cluster_instance_endpoints" {
  description = "A list of all cluster instance endpoints"
  value       = aws_rds_cluster_instance.this.*.endpoint
}
```

Code styling

- Examples and Terraform modules should contain documentation explaining features and how to use them.
- All links in README.md files should be absolute to make Terraform Registry website show them correctly.
- Documentation may include diagrams created with mermaid and blueprints created with cloudcraft.co.
- Use Terraform pre-commit hooks to make sure that the code is valid, properly formatted, and automatically documented before it is pushed to git and reviewed by humans.

Documentation

Automatically generated documentation

pre-commit is a framework for managing and maintaining multi-language pre-commit hooks. It is written in Python and is a powerful tool to do something automatically on a developer's machine before code is committed to a git repository. Normally, it is used to run linters and format code (see supported hooks).

With Terraform configurations `pre-commit` can be used to format and validate code, as well as to update documentation.

Check out the pre-commit-terraform repository to familiarize yourself with it, and existing repositories (eg, terraform-aws-vpc) where this is used already.

terraform-docs

terraform-docs is a tool that does the generation of documentation from Terraform modules in various output formats. You can run it manually (without pre-commit hooks), or use pre-commit-terraform hooks to get the documentation updated automatically.

@todo: Document module versions, release, GH actions

Resources

1. pre-commit framework homepage
2. Collection of git hooks for Terraform to be used with pre-commit framework
3. Blog post by Dean Wilson: pre-commit hooks and terraform - a safety net for your repositories

FAQ

FTP (Frequent Terraform Problems)

What are the tools I should be aware of and consider using?

- **Terragrunt** - Orchestration tool
- **tflint** - Code linter
- **tfenv** - Version manager
- **Atlantis** - Pull Request automation
- **pre-commit-terraform** - Collection of git hooks for Terraform to be used with pre-commit framework
- **Infracost** - Cloud cost estimates for Terraform in pull requests. Works with Terragrunt, Atlantis and pre-commit-terraform too.

Have you had a chance to answer the previous question?

Yes, after a few months we finally found the answer. Sadly, Mike is on vacations right now so I'm afraid we are not able to provide the answer at this point.

What are the solutions to dependency hell with modules?

Versions of resource and infrastructure modules should be specified. Providers should be configured outside of modules, but only in composition. Version of providers and Terraform can be locked also.

There is no master dependency management tool, but there are some tips to make dependency specifications less problematic. For example, Dependabot can be used to automate dependency updates. Dependabot creates pull requests to keep your dependencies secure and up-to-date. Dependabot supports Terraform configurations.

References

> There are a lot of people who create great content and manage open-source projects relevant to the Terraform community but I can't think of the best structure to get these links listed here without copying lists like awesome-terraform.

https://twitter.com/antonbabenko/lists/terraform-experts - List of people who work with Terraform very actively and can tell you a lot (if you ask them).

https://github.com/shuaibiyy/awesome-terraform - Curated list of resources on HashiCorp's Terraform.

http://bit.ly/terraform-youtube - "Your Weekly Dose of Terraform" YouTube channel by Anton Babenko. Live streams with reviews, interviews, Q&A, live coding, and some hacking with Terraform.

https://weekly.tf - Terraform Weekly newsletter. Various news in the Terraform world (projects, announcements, discussions) by Anton Babenko.

Writing Terraform configurations

Use `locals` to specify explicit dependencies between resources

Helpful way to give a hint to Terraform that some resources should be deleted before even when there is no direct dependency in Terraform configurations.

https://raw.githubusercontent.com/antonbabenko/terraform-best-practices/master/snippets/locals.tf

Terraform 0.12 - Required vs Optional arguments

1. Required argument `index_document` must be set, if `var.website` is not an empty map.
2. Optional argument `error_document` can be omitted.

main.tf

```
variable "website" {
  type    = map(string)
  default = {}
}

resource "aws_s3_bucket" "this" {
  # omitted...

  dynamic "website" {
    for_each = length(keys(var.website)) == 0 ? [] : [var.website]

    content {
      index_document = website.value.index_document
      error_document = lookup(website.value, "error_document", null)
    }
  }
}
```

terraform.tfvars

```
website = {
  index_document = "index.html"
}
```

Workshop

There is also a workshop for people who want to practice some of the things described in this guide.

The content is here - https://github.com/antonbabenko/terraform-best-practices-workshop