TERRAFORM

What Are Terraform Modules and How to Use Them: Tutorial

Stanislaw Szymanski

Updated 12 Oct 2023 · 12 min read

Terraform makes it easier to grow your infrastructure and keep its configuration clean. But, as the infrastructure grows, a single directory becomes hard to manage.

That's where Terraform modules come in.

In this post you will learn what Terraform modules are, how to use them, and what problems they solve.

What Is a Terraform Module?

A *Terraform module* is a collection of standard configuration files in a dedicated directory. Terraform modules encapsulate groups of resources dedicated to one task, reducing the amount of code you have to develop for similar infrastructure components.

Some say that Terraform modules are a way of extending your present Terraform configuration with existing parts of reusable code reducing the amount of code you have to develop for similar infrastructure components. Others say that the Terraform module definition is a single or many .tf files stacked together in their own directory. Both are correct.

Module blocks can also be used to force compliance on other resources—to deploy databases with encrypted disks, for example. By hard-coding the encryption configuration and not exposing it through variables, you're making sure that every time the module is used, the disks are going to be encrypted.

A typical module can look like this:

```
.
├── main.tf
├── outputs.tf
├── README.md
```


ule in itself. If you

ered a **root module**.

If you need more help learning about Terraform, check out our Getting Started With Terraform on AWS tutorial.

Note: New versions of Terraform will be placed under the BUSL license, but everything created before version 1.5.x stays open-source. OpenTofu is an open-source version of Terraform that will expand on Terraform's existing concepts and offerings. It is a viable alternative to HashiCorp's Terraform, being forked from Terraform version 1.5.6. OpenTofu retained all the features and functionalities that had made Terraform popular among developers while also introducing improvements and enhancements. OpenTofu is not going to have its own providers and modules, but it is going to use its own registry for them.

How To Use Terraform Modules

OK, now that you know what a Terraform module is, let's take a look at a step-by-step process of creating them. So, let's get started!

1. Declare that you want to use Terraform modules

To use a Terraform module, you have to first declare that you wish to use it in your current configuration. To do this, use the module block and provide the appropriate variable values:

```
module "terraform_test_module" {
```

As you're adding the variables, there are a few arguments that you should keep an eye on: their source, version, and four meta-arguments.

Sources

Terraforms modules can be stored either **locally or remotely**. The `source` argument will change depending on their location. For example, if the module you wish to call is stored in a directory named "terraform-test-module" located in the same place as your root module directory, your root configuration would look like this:

```
module "terraform_test_module" {
 source  = "./terraform-test-module"
[...]
```

But, if you want to keep the module in a VCS (let's assume git and GitHub), the source will need to point to the correct repository containing the appropriate version of the module that you're calling:

```
module "terraform_test_module" {
 source  = "git@github.com:your-organization/the-repository-name.git?ref=1.0.0"
[...]
```


are places where
u store the ones you

```
module "terraform_test_module" {
 source  = "<registry address/<organization>/<provider>/<module name>"
 version = "1.0.0"
[...]
```

Please note that the URL scheme depends on the registry provider. For example, imagine that you wish to include a VPC module stored in the Spacelift registry in your configuration:

```
module "vpc" {
 source  = "spacelift.io/your-organization/vpc-module-name/aws"
 version = "1.0.0"
[...]
```

As you've probably noticed, those two examples are a bit different from one another. A big advantage of the Spacelift's registry is that it provides an example for each module. This means that you can check whether the source argument in your configuration is correct. Additionally, you can grab the entire example and use it as a template, filling in the blanks, and starting your work right away!

Versions

Versioning enables you to control what module changes should be introduced into your
rastructure caused
ience that any
go wrong. This can

The first part is an operator:

- "=" (or no operator) means "Only one version—this specific version."
- "!=" translates to "Other versions are fine, except this one here"
- ">, >=, <, <=" are used for comparisons. For example, "Use any version newer than 1.0.0, but older than 1.1.0"
- "~>" is quite interesting. This operator allows only the rightmost part of the version number to increment. In other words, "~>2.6.0" would mean that you wish to use the newest patch version of the module (2.6.<anything>), but not the newer minor or major versions (2.7.0 or above).

The second part of the syntax is the version number. Although it isn't absolutely required (except when using registries), it's best to stick to the Semantic Versioning convention—it's easy to understand and very transparent. One look at the version, and you know where you are.

As you might have already noticed, there is an **exception** to the "keep things versioned" rule—in two of the examples above, the version argument is defined. In one, it's not. The reason for this is that Terraform considers modules loaded from the same source repository as being of the equal version to the caller. Local modules, by the way, need no version constraints whatsoever.

Meta-Arguments

Meta-arguments are special arguments that change the behavior of Terraform when parsing

accounts and you want to create resources bound to the secondary. In order to achieve that, you need to properly design your Terraform module so that it takes the provider as an argument and passes it along, like this:

```
provider "aws" {
 alias  = "frankfurt"
 region = "eu-central-1"
}

module "example" {
[...]

 providers = {
   aws.nested_provider_alias = aws.frankfurt
 }
}
```

- "depends_on"—usually, Terraform handles implicit dependencies quite well, but sometimes they aren't enough. If you need to declare that something should be created before something else, use this meta-argument to define an explicit boundary between resources or modules.

💡 **You might also like:**

2. Declare Module Outputs

Sometimes you might need to use the values that are available in the already created resources. A Terraform module completely encapsulates those resources, and here's how they can be accessed.

First, declare in your Terraform module that the selected value should be available as an output:

```
output "random_string" {
 value       = aws_example_resource.example_device.random_string
 description = "A random string from an example resource on AWS."
}
```

Then, call this value like this:

```
resource "example_resource" "example" {
 [...]

 random_string = module.example.random_string
}
```

need to manually taint every resource that was created by the module. While it may seem like a downside, it is actually a feature that offers **additional protection** against human error. What's more, you will rarely need to taint an entire Terraform module.

To taint the resources that were created by the example module presented above, try this:

terraform taint module.example.aws_example_resource.example_something

4. Test Modules

Whether a blessing or a curse, Terraform's code is still code, and should be properly tested. It is absolutely crucial. But when done manually, it can be quite a chore. Luckily, Spacelift, alongside other upgrades and extensions, provides **automated module testing** and you will absolutely love it.

Terraform modules managed by the Spacelift registry get tested every time you push a new commit. They are applied, deleted, and you get all of the details on the run. If something fails along the way, the entire test fails, too. If everything goes smoothly, the test is successful. The entire process is automatic, but if you feel like playing around with it, you can even write your own tests, such as this one:

```
provider "aws" {
 region = "eu-central-1"
}

resource "random_string" "this" {
```

```
module "test" {
 source = "../"

 cidr_blocks = {
   eu-central-1a = "10.10.0.0/24"
 }

 environment = "test-${random_string.this.result}"
 name        = "test-${random_string.this.result}"
 vpc_id      = aws_vpc.test.id
}
```

Customizable automated testing—what's not to love here?

What Problems Do Terraform Modules Solve?

Alright, now that you know how to use Terraform modules and what to keep an eye on when doing it, let's see what problems Terraform modules solve.

1. Code Repetition

As your Terraform infrastructure grows in scale, the code will need to do so as well. Copy-pasting an entire stack of code whenever you need more than a single instance of something isn't scalable or efficient. It's a waste of many things, time in particular.

ocuments itself. A

connecting modules

When you create a Terraform module in compliance with appropriate standards and best practices, whenever you reuse it, it follows the same right pattern. No matter if it's encryption, redundancy, or lifecycle policies—practices configured inside the module will be enforced, so that you won't have to do it again personally.

4. Human Error

When you copy-paste or create a group of resources from scratch, it's easy to make a mistake, like rewriting or renaming something. To make sure that doesn't happen, create a single Terraform module, test it, then use it in multiple places to check whether all of the elements are correct. It's easier to check and test what you type into a single block than scroll, jumping from one place to another, and so on.

As you can see, there are a lot of advantages to using Terraform modules, so long as they aren't overused. Find the right balance and keep it.

Terraform Modules Best Practices

- Each Terraform module should live in its own repository and versioning should be leveraged.
- Minimal structure: main.tf, variables.tf, outputs.tf.
- Each Terraform module should have examples inside of them.
- Use input and output variables (outputs can be accessed with `module.module_name.output_name`).

Key Points

The logic of Terraform modules may seem a bit complicated at first, but in the end, is a simple and efficient way of building your infrastructure quickly and reliably. If you want to step up your efforts, check out Spacelift and the additional features it offers, which extend and perfect the modular approach.

Spacelift provides everything you need to make your module easily maintainable and usable. There is CI/CD for multiple specified versions of Terraform, which is capable of testing your module on each commit. You get an autogenerated page describing your Module and its intricacies, so your users can explore them and gather required information at a glimpse. It's also deeply integrated with all the features Stacks use which you know and love, like Environments, Policies, Contexts, and Worker Pools. Create a trial account and check it out for free.

Cross-organization private module sharing and the above-mentioned module testing automation will help you get the most out of your configuration and make its development a lot smoother.

If you need more help with Terraform, I encourage you to check the following blog posts: How to Automate Terraform Deployments, and 12 Terraform Best Practices.

Enjoy writing your code and growing your infrastructure, and have a productive day!

Written by

Stanislaw Szymanski

Stanislaw is a guest writer at Spacelift. He is a DevOps Engineer at Paragon DCX. He holds HashiCorp Certified: Terraform Associate certification. IT handyman - been there, done that... From hardware diagnostics and repair, through own web-hosting, to the intricacies of the clouds.

TERRAFORM

9 min read

Terraform Taint, Untaint, Replace – How to Use It (Examples)

TERRAFORM

18 min read

How to Use Terraform Variables (Locals, Input, Output, Environment)

spacelift

Product **Company** **Learn**

Terraform Modules Cheatsheet

Grab our ultimate cheat sheet PDF to master building reusable Terraform modules!

ernative

loud Alternative

nterprise Alternative

Integrations

Terraform Automation

Security

System Status

Product Updates

Get our newsletter

Subscribe

Privacy Policy Terms of Service

Terraform Modules Cheatsheet

Grab our ultimate cheat sheet PDF to master building reusable Terraform modules!