



Search Terraform documentation

Creating Modules

Hands-on: Try the [Reuse Configuration with Modules](#) collection on HashiCorp Learn.

A *module* is a container for multiple resources that are used together. Modules can be used to create lightweight abstractions, so that you can describe your infrastructure in terms of its architecture, rather than directly in terms of physical objects.

The `.tf` files in your working directory when you run `terraform plan` or `terraform apply` together form the *root* module. That module may [call other modules](#) and connect them together by passing output values from one to input values of another.

To learn how to *use* modules, see [the Modules configuration section](#). This section is about *creating* re-usable modules that other configurations can include using `module` blocks.

Module structure

Re-usable modules are defined using all of the same [configuration language](#) concepts we use in root modules. Most commonly, modules use:

- [Input variables](#) to accept values from the calling module.
- [Output values](#) to return results to the calling module, which it can then use to populate arguments elsewhere.
- [Resources](#) to define one or more infrastructure objects that the module will manage.

To define a module, create a new directory for it and place one or more `.tf` files inside just as you would do for a root module. Terraform can load modules either from local relative paths or from remote repositories; if a module will be re-used by lots of configurations you may wish to place it in its own version control repository.

Modules can also call other modules using a `module` block, but we recommend keeping the module tree relatively flat and using [module composition](#) as an alternative to a deeply-nested tree of modules, because this makes the individual modules easier to re-use in different combinations.

When to write a module

In principle any combination of resources and other constructs can be factored out into a module, but over-using modules can make your overall Terraform configuration harder to understand and maintain, so we recommend moderation.

A good module should raise the level of abstraction by describing a new concept in your architecture that is constructed from resource types offered by providers.


For example, `aws_instance` and `aws_elb` are both resource types

belonging to the AWS provider. You might use a module to represent the higher-level concept "HashiCorp Consul cluster running in AWS" which happens to be constructed from these and other AWS provider resources.

We *do not* recommend writing modules that are just thin wrappers around single other resource types. If you have trouble finding a name for your module that isn't the same as the main resource type inside it, that may be a sign that your module is not creating any new abstraction and so the module is adding unnecessary complexity. Just use the resource type directly in the calling module instead.

Refactoring module resources

You can include [refactoring blocks](#) to record how resource names and module structure have changed from previous module versions. Terraform uses that information during planning to reinterpret existing objects as if they had been created at the corresponding new addresses, eliminating a separate workflow step to replace or migrate existing objects.

 [Edit this page](#)

[Overview](#)

[Docs](#)

[Extend](#)

[Privacy](#)

[Security](#)

[Press Kit](#)

[Consent Manager](#)