Providers Within Modules

JUMP TO SECTION

In a configuration with multiple modules, there are some special considerations for how resources are associated with provider configurations.

Each resource in the configuration must be associated with one provider configuration. Provider configurations, unlike most other concepts in Terraform, are global to an entire Terraform configuration and can be shared across module boundaries. Provider configurations can be defined only in a root Terraform module.

Providers can be passed down to descendent modules in two ways: either *implicitly* through inheritance, or *explicitly* via the `providers` argument within a `module` block. These two options are discussed in more detail in the following sections.

A module intended to be called by one or more other modules must not contain any `provider` blocks. A module containing its own provider configurations is not compatible with the `for_each`, `count`, and `depends_on` arguments that were introduced in Terraform v0.13. For more information, see Legacy Shared Modules with Provider Configurations.

Provider configurations are used for all operations on associated resources, including destroying remote objects and refreshing state. Terraform retains, as part of its state, a reference to the provider configuration that was most recently used to apply changes to each resource. When a `resource` block is removed from the configuration, this record in the state will be used to locate the appropriate configuration because the resource's `provider` argument (if any) will no longer be present in the configuration.

As a consequence, you must ensure that all resources that belong to a particular provider configuration are destroyed before you can remove that provider configuration's block from your configuration. If Terraform finds a resource instance tracked in the state whose provider configuration block is no longer available then it will return an error during planning, prompting you to reintroduce the provider configuration.

Provider Version Constraints in Modules

Although provider *configurations* are shared between modules, each module must declare its own provider requirements (/docs/language/providers/requirements.html), so that Terraform can ensure that there is a single version of the provider that is compatible with all modules in the configuration and to specify the source address (/docs/language/providers/requirements.html#source-addresses) that serves as the global (module-agnostic) identifier for a provider.

To declare that a module requires particular versions of a specific provider, use a `required_providers` block inside a `terraform` block:

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = ">= 2.7.0"
    }
  }
}
```

A provider requirement says, for example, "This module requires version v2.7.0 of the provider `hashicorp/aws` and will refer to it as `aws`." It doesn't, however, specify any of the configuration settings that determine what remote endpoints the provider will access, such as an AWS region; configuration settings come from provider *configurations*, and a particular overall Terraform configuration can potentially have several different configurations for the same provider (/docs/language/providers/configuration.html#alias-multiple-provider-configurations).

Provider Aliases Within Modules

To declare multiple configuration names for a provider within a module, add the `configuration_aliases` argument:

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = ">= 2.7.0"
      configuration_aliases = [ aws.alternate ]
    }
  }
}
```

The above requirements are identical to the previous, with the addition of the alias provider configuration name `aws.alternate`, which can be referenced by resources using the `provider` argument.

If you are writing a shared Terraform module, constrain only the minimum required provider version using a `>=` constraint. This should specify the minimum version containing the features your module relies on, and thus allow a user of your module to potentially select a newer provider version if other features are needed by other parts of their overall configuration.

Implicit Provider Inheritance

For convenience in simple configurations, a child module automatically inherits default (un-aliased) provider configurations from its parent. This means that explicit `provider` blocks appear only in the root module, and downstream modules can simply declare resources for that provider and have them automatically associated with the root provider configurations.

For example, the root module might contain only a `provider` block and a `module` block to instantiate a child module:

```
provider "aws" {
  region = "us-west-1"
}

module "child" {
  source = "./child"
}
```

The child module can then use any resource from this provider with no further provider configuration required:

```
resource "aws_s3_bucket" "example" {
  bucket = "provider-inherit-example"
}
```

We recommend using this approach when a single configuration for each provider is sufficient for an entire configuration.

Note: Only provider configurations are inherited by child modules, not provider source or version requirements. Each module must declare its own provider requirements (/docs/language/providers/requirements.html). This is especially important for non-HashiCorp providers.

In more complex situations there may be multiple provider configurations (/docs/language/providers/configuration.html#alias-multiple-provider-configurations), or a child module may need to use different provider settings than its parent. For such situations, you must pass providers explicitly.

Passing Providers Explicitly

When child modules each need a different configuration of a particular provider, or where the child module requires a different provider configuration than its parent, you can use the `providers` argument within a `module` block to explicitly define which provider configurations are available to the child module. For example:

```
# The default "aws" configuration is used for AWS resources in the root
# module where no explicit provider instance is selected.
provider "aws" {
  region = "us-west-1"
}

# An alternate configuration is also defined for a different
# region, using the alias "usw2".
provider "aws" {
  alias  = "usw2"
  region = "us-west-2"
}

# An example child module is instantiated with the alternate configuration,
# so any AWS resources it defines will use the us-west-2 region.
module "example" {
  source    = "./example"
  providers = {
    aws = aws.usw2
  }
}
```

The `providers` argument within a `module` block is similar to the `provider` argument (/docs/language/meta-arguments/resource-provider.html) within a resource, but is a map rather than a single string because a module may contain resources from many different providers.

The keys of the `providers` map are provider configuration names as expected by the child module, and the values are the names of corresponding configurations in the *current* module.

Once the `providers` argument is used in a `module` block, it overrides all of the default inheritance behavior, so it is necessary to enumerate mappings for *all* of the required providers. This is to avoid confusion and surprises that may result when mixing both implicit and explicit provider passing.

Additional provider configurations (those with the `alias` argument set) are *never* inherited automatically by child modules, and so must always be passed explicitly using the `providers` map. For example, a module that configures connectivity between

networks in two AWS regions is likely to need both a source and a destination region. In that case, the root module may look something like this:

```
provider "aws" {
  alias  = "usw1"
  region = "us-west-1"
}

provider "aws" {
  alias  = "usw2"
  region = "us-west-2"
}

module "tunnel" {
  source    = "./tunnel"
  providers = {
    aws.src = aws.usw1
    aws.dst = aws.usw2
  }
}
```

The subdirectory `./tunnel` must then declare the configuration aliases for the provider so the calling module can pass configurations with these names in its `providers` argument:

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = ">= 2.7.0"
      configuration_aliases = [ aws.src, aws.dest ]
    }
  }
}
```

Each resource should then have its own `provider` attribute set to either `aws.src` or `aws.dst` to choose which of the two provider configurations to use.

Legacy Shared Modules with Provider Configurations

In Terraform v0.10 and earlier there was no explicit way to use different configurations of a provider in different modules in the same configuration, and so module authors commonly worked around this by writing `provider` blocks directly inside their modules, making the module have its own separate provider configurations separate from those declared in the root module.

However, that pattern had a significant drawback: because a provider configuration is required to destroy the remote object associated with a resource instance as well as to create or update it, a provider configuration must always stay present in the overall Terraform configuration for longer than all of the resources it manages. If a particular module includes both resources and the provider configurations for those resources then removing the module from its caller would violate that constraint: both the resources and their associated providers would, in effect, be removed simultaneously.

Terraform v0.11 introduced the mechanisms described in earlier sections to allow passing provider configurations between modules in a structured way, and thus we explicitly recommended against writing a child module with its own provider configuration blocks. However, that legacy pattern continued to work for compatibility purposes -- though with the same drawback -- until Terraform v0.13.

Terraform v0.13 introduced the possibility for a module itself to use the `for_each`, `count`, and `depends_on` arguments, but the implementation of those unfortunately conflicted with the support for the legacy pattern.

To retain the backward compatibility as much as possible, Terraform v0.13 continues to support the legacy pattern for module blocks that do not use these new features, but a module with its own provider configurations is not compatible with `for_each`, `count`, or `depends_on`. Terraform will produce an error if you attempt to combine these features. For example:

```
Error: Module does not support count

  on main.tf line 15, in module "child":
  15:   count = 2

Module "child" cannot be used with count because it contains a nested provider
configuration for "aws", at child/main.tf:2,10-15.

This module can be made compatible with count by changing it to receive all of
its provider configurations from the calling module, by using the "providers"
argument in the calling module block.
```

To make a module compatible with the new features, you must remove all of the `provider` blocks from its definition.

If the new version of the module declares `configuration_aliases`, or if the calling module needs the child module to use different provider configurations than its own default provider configurations, the calling module must then include an explicit `providers` argument to describe which provider configurations the child module will use:

```
provider "aws" {
  region = "us-west-1"
}

provider "aws" {
  region = "us-east-1"
  alias  = "east"
}

module "child" {
  count = 2
  providers = {
    # By default, the child module would use the
    # default (unaliased) AWS provider configuration
    # using us-west-1, but this will override it
    # to use the additional "east" configuration
    # for its resources instead.
    aws = aws.east
  }
}
```

Since the association between resources and provider configurations is static, module calls using `for_each` or `count` cannot pass different provider configurations to different instances. If you need different instances of your module to use different provider configurations then you must use a separate `module` block for each distinct set of provider configurations:

```
provider "aws" {
  alias  = "usw1"
  region = "us-west-1"
}

provider "aws" {
  alias  = "usw2"
  region = "us-west-2"
}

provider "google" {
  alias       = "usw1"
  credentials = "${file("account.json")}"
  project     = "my-project-id"
  region      = "us-west1"
  zone        = "us-west1-a"
}

provider "google" {
  alias       = "usw2"
  credentials = "${file("account.json")}"
  project     = "my-project-id"
  region      = "us-west2"
  zone        = "us-west2-a"
}

module "bucket_w1" {
  source    = "./publish_bucket"
  providers = {
    aws.src    = aws.usw1
    google.src = google.usw2
  }
}

module "bucket_w2" {
  source    = "./publish_bucket"
  providers = {
    aws.src    = aws.usw2
    google.src = google.usw2
  }
}
```