Creating Module Dependencies in Terraform 0.13

Roger Berlind
Jun 22, 2020 · 7 min read ★

Introduction

Terraform 0.13 adds a number of new features including improved usability of modules, automated installation of third-party providers, and custom validation of variable values.

Improved usability of modules is delivered in two ways:

1. Users can now instantiate multiple instances of a single module block with the `for_each` and `count` meta-arguments just as they can currently do for resources in Terraform 0.12.
2. Users can now declare that a module depends on one or more external resources or even other modules with the `depends_on` meta-argument just as they can currently do for resources in Terraform 0.12.

This blog post discusses an example of making one module depend on another module with the new `depends_on` meta-argument. This has been one of the most frequently

requested enhancements in Terraform for a long time.

Users interested in an example of using the `for_each` meta-argument with a module can see this example or this one from the docs.

Prior to Terraform 0.13, module instances only served as namespaces for collections of resources and data sources. They were not nodes within Terraform's dependency graph. While it was possible in Terraform 0.12 to make a resource depend on a module, you could not make an entire module depend on a resource or on an entire module.

Terraform 0.13 addresses this by adding the new `depends_on` argument to the `module` blocks within your Terraform code.

An Overview of the Example

In this example, we're going to use the local_file resource in one module, write-files, to write 3 files (apple.txt, banana.txt, and orange.txt) to the root module's directory and then use the local_file data source in a second module, read-files, to read the files and output their contents.

We will also use more complicated versions of these modules, write-files-complicated and read-files-complicated, to show how module dependency could be faked in older versions of Terraform without the `depends_on` argument of `module` blocks.

This is a useful (although trivial) example of using the `depends_on` argument of `module` blocks because the data sources that read the files have no direct dependency on the resources that write them. Obviously, we want Terraform to write the files in the first module before it tries to read them in the second module. Otherwise, we'll either get errors or read back empty files before their contents have been written.

We'll actually describe 3 variations of this example:

1. A naive "Hope for the Best" approach that does not set any explicit dependencies. (This uses main-1.tf.)
2. A "Fake Module Dependencies" approach that works but is complicated. (This uses main-2.tf.)

3. A “True Module Dependencies” approach that uses the new `depends_on` argument in the second module’s block. (This uses main-3.tf)

The README.md associated with this example provides complete instructions for running all 3 variations. So, we won’t describe all the steps or all the various results here.

The Simple Versions of the Modules

The main.tf file of the write-files module has this code:

```
resource "local_file" "apple" {
    content     = "apple"
    filename = "${path.root}/apple.txt"
}

resource "local_file" "banana" {
    content     = "banana"
    filename = "${path.root}/banana.txt"
}

resource "local_file" "orange" {
    content     = "orange"
    filename = "${path.root}/orange.txt"
}
```

It will create 3 files named after 3 different fruits. Each of these files will have the name of the fruit.

The main.tf file of the read-files module has this code:

```
data "local_file" "apple" {
    filename = "${path.root}/apple.txt"
}

data "local_file" "banana" {
    filename = "${path.root}/banana.txt"
}

data "local_file" "orange" {
    filename = "${path.root}/orange.txt"
}

output "fruit" {
  value = [
    data.local_file.apple.content,
```

```
    data.local_file.banana.content,
    data.local_file.orange.content,
  ]
}
```

It will read the 3 files and then combine their content in an output of type list. When the code works as desired, the output should show this:

```
fruit = [
  "apple",
  "banana",
  "orange",
]
```

The Hope for the Best Approach

The main-1.tf file has the following code:

```
module "write-files" {
  source = "./modules/write-files"
}

module "read-files" {
  source = "./modules/read-files"
}

output "fruit" {
  value = module.read-files.fruit
}
```

It references the write-files and read-files modules and then tries to generate an output from the latter. Note that it does not set any explicit dependencies between the modules.

If you run `terraform init` and `terraform plan`, you will get errors indicating that the fruit files do not exist. The plan cannot read the files because they do not yet exist.

We can partially address this problem by creating empty versions of the files before doing a plan or apply. The plan will then work correctly, but if you run `terraform apply`, you get the following output:

```
fruit = [
  "",
  "",
  "",
]
```

The problem is that the files were actually read by the plan that the apply triggered which means the empty files were read. So, the "Hope for the Best" approach fails.

The Fake Module Dependencies Approach

The main-2.tf file has the following code:

```
module "write-files-complicated" {
  source = "./modules/write-files-complicated"
}

module "read-files-complicated" {
  source = "./modules/read-files-complicated"
  wait_for_write = module.write-files-complicated.write_done
}

output "fruit" {
  value = module.read-files-complicated.fruit
}
```

Note that we are using the write-files-complicated and read-files-complicated modules instead of the original write-files and read-files modules. Additionally, we are setting the `wait_for_write` variable of the read-files-complicated module to the `write_done` output of the write-files-complicated module. It is the code in the modified modules and the passing of the `write_done` output into the `wait_for_write` variable that fakes the dependency between the modules.

When we say that this approach fakes the dependency between the modules, we mean that it artificially creates a dependency for each `local_file` data source of the read-files-complicated module on an extraneous null resource that reads the `wait_for_write` variable that is set to the `write_done` output. The code below will make this more clear.

The main.tf file of the write-files-complicated module has the same code as the write-files module except that it adds this output:

```
output "write_done" {
  value = "apples, bananas, and oranges"
}
```

The main.tf file of the read-files-complicated module is more complicated, having this code:

```
variable "wait_for_write" {}

resource "null_resource" "dependency" {
  triggers = {
    dependency_id = var.wait_for_write
  }

}

data "local_file" "apple" {
    filename = "${path.root}/apple.txt"
    depends_on = [null_resource.dependency]
}

data "local_file" "banana" {
    filename = "${path.root}/banana.txt"
    depends_on = [null_resource.dependency]
}

data "local_file" "orange" {
    filename = "${path.root}/orange.txt"
    depends_on = [null_resource.dependency]
}

output "fruit" {
  value = [
    data.local_file.apple.content,
    data.local_file.banana.content,
    data.local_file.orange.content,
  ]
}
```

Note the following changes:

1. It adds the `wait_for_write` variable.
2. It adds a null resource that references the variable.

3. Each of the `local_file` data sources has a `depends_on` argument that references the null resource. These things were added to make sure that none of the data sources could be created and read before the variable is given a value and the null resource is created. And that will not be possible until after the write-files-complicated module has finished writing the files.

Running `terraform init` and `terraform destroy` (after renaming some files) does give the following desired output:

```
fruit = [
  "apple",
  "banana",
  "orange",
]
```

In this approach, everything works correctly because the reading of the files is delayed until the apply (instead of being done during the plan) and is even delayed until after the files have been written.

However, this approach is quite awkward because we had to add an extraneous variable, an extraneous output, an extraneous null resource and 3 `depends_on` arguments.

The True Module Dependencies Approach

Finally, we will now show the use of the new `depends_on` meta-argument of the `module` block in our third "True Module Dependencies" approach.

The main-3.tf file has the following code:

```
module "write-files" {
  source = "./modules/write-files"
}

module "read-files" {
  source = "./modules/read-files"
  depends_on = [module.write-files]
}

output "fruit" {
  value = module.read-files.fruit
}
```

Note the use of the new `depends_on` argument in the second module block. Also note that we have reverted to using the simpler write-files and read-files modules.

When you run `terraform apply` and type "yes" to confirm that you rally want to apply, you'll see the following output:

```
module.write-files.local_file.apple: Creating...
module.write-files.local_file.orange: Creating...
module.write-files.local_file.banana: Creating...
module.write-files.local_file.banana: Creation complete after 0s
[id=250e77f12a5ab6972a0895d290c4792f0a326ea8]
module.write-files.local_file.orange: Creation complete after 0s
[id=ef0ebbb77298e1fbd81f756a4efc35b977c93dae]
module.write-files.local_file.apple: Creation complete after 0s
[id=d0be2dc421be4fcd0172e5afceea3970e2f3d940]
module.read-files.data.local_file.apple: Reading...
module.read-files.data.local_file.orange: Reading...
module.read-files.data.local_file.banana: Reading...
module.read-files.data.local_file.orange: Read complete after 0s
[id=ef0ebbb77298e1fbd81f756a4efc35b977c93dae]
module.read-files.data.local_file.apple: Read complete after 0s
[id=d0be2dc421be4fcd0172e5afceea3970e2f3d940]
module.read-files.data.local_file.banana: Read complete after 0s
[id=250e77f12a5ab6972a0895d290c4792f0a326ea8]

Apply complete! Resources: 3 added, 0 changed, 0 destroyed.

Outputs:

fruit = [
  "apple",
  "banana",
  "orange",
]
```

Note that the reads were done after the writes and that the output shows the names of the 3 fruits as desired.

While the result is quite similar to our second approach, the code is much less complex. We did not need to create extraneous objects and only needed one `depends_on` argument, this time at the module level instead of on each data source of the second module.

Running the Example Yourself

The README.md associated with the terraform-0.13-examples directory of the hashicorp/terraform-guides repository example provides instructions for downloading

and deploying the terraform 0.13.0 binary. The README.md associated with this example provides complete instructions for running all 3 variations of the example.

Conclusion

This blog post and the associated example have demonstrated how much easier it is to create dependencies between modules with the new `depends_on` meta-argument of the `module` block in Terraform 0.13. While the example is trivial, we hope you will find the code useful in your own Terraform projects that require explicit module dependencies in more serious use cases.

Hashicorp Terraform Modules Dependencies 13