spacelift

// The Most Flexible IaC Platform

# Terraform Modules Cheatsheet

Modules are the building blocks for achieving more complex infrastructure, making your code reusable

- A reusable collection of Terraform resources defined together and managed as a group.
- You can think of modules as blueprints
- Whenever you are adding a new module to a configuration, you have to run terraform init in order to create the .terraform directory and the .terraform.lock.hcl file; this will download the module (even for local sources) and the required providers, setting also the versions in the lock file
- All variables defined inside a module become parameters of the child objects created for that module (some of them will have default values, so you won't be required to specify them)

## Supported Modules Sources

- The source argument inside of the module block tells Terraform from which location it can get the module code
- The version argument inside the module block should be used with module registries

### Local Source

- Uses a path to a folder containing the configuration of the module

```
module "module1" {
 source = "./module1"
}
```

### Terraform registry

- The module name should be referenced in the following form: <NAMESPACE>/<NAME>/<PROVIDER>

```
module "ec2-instance" {
 source  = "terraform-aws-modules/ec2-instance/aws"
 version = "4.3.0"
}
```

### Spacelift Registry

- The name of the module should be referenced in the following form spacelift.io/<organization>/<module_name>/<provider>

```
module "module1" {
 source  = "spacelift.io/spacelift-io/module1/aws"
 version = "1.0.1"
}
```

### GitHub

- Unprefixed github.com urls will be recognized by Terraform
- Similarly, the same is true for git@github.com

```
# HTTP
module "ec2_http" {
 source = "github.com/user/ec2"
}

# SSH
module "ec2_ssh" {
 source = "git@github.com:user/ec2.git"
}
```

### BitBucket

- Unprefixed bitbucket.org urls will be recognized by Terraform

```
module "module1" {
 source = "bitbucket.org/user/module1"
}
```

### HTTP

- Terraform sends a GET request and gets the module from the http/https url
- Archives can also be used (supported types: zip, tar.bz2, tar.gz, tar.xz)

```
module "module1" {
 source = "https://example.com/modules/module1"
}
```

```
module "module1" {
 source = "https://example.com/module1?archive=zip"
}
```

### Generic Git Repository

- Other git repositories will be recognized if you prefix them with "git::"

```
# HTTP
module "ec2_http" {
 source = "git::https://example.com/ec2.git"
}

# SSH
module "ec2_ssh" {
 source = "git::ssh://username@example.com/ec2.git"
}
```

### Selecting Revisions

- By default, the main branch will be selected, but you can use any tag, commit sha or branch to get a module

```
# branch
module "module1" {
 source = "github.com/user/module1?ref=dev"
}

# tag
module "module1" {
 source = "github.com/user/module1?ref=v1.0.1"
}

# commit sha
module "module1" {
 source = "github.com/user/module1?ref=588c6c..."
}
```

### S3 Bucket

- Archives stored in S3 can be used with the "s3::" prefix

```
module "module1" {
 source = "s3::https://s3-eu-west-1.amazonaws.com/terraform-modules/module1.zip"
}
```

## Meta-Arguments

### Depends_on

- Establishes dependencies between a module and another component (even another module)
- Works the same as depends_on would work on any type of resource/data source/local

```
module "module1" {
 depends_on = [
  null_resource.this
 ]
 source = "./module1"
}

resource "null_resource" "this" {
 provisioner "local-exec" {
  command = "echo a"
 }
}

# The depends_on inside module1, ensures the null resource is created before the module.
```

### Providers

- The provider block inside a module is pretty different than the ones inside of another component, mainly because it is a different data type (map vs string)
- If there is only one provider inside the configuration, the providers block is not necessary as modules will inherit that single provider
- However, if there are multiple providers inside the configuration and some of the same type (2 aws providers, one defined with an alias, for example), if you want to use the one with the alias, you will need to explicitly specify it inside the providers block

```
module "module1" {
 source = "./module1"
 providers = {
  aws = aws.eastus1
 }
}


# If your module requires multiple providers, you can define all of them in providers block similar to the below code

module "module1" {
 source = "./module1"
 providers = {
  aws.source    = aws.use1
  aws.destination = aws.use2
 }
}
```

### For_each

- Used as you would use it on any other component
- Create multiple instances of the same module

```
module "module1" {
 source = "./module1"
 for_each = {
  key  = "value"
  key2 = "value2"
 }
}

# module.module1["key"]... will be created
# module.module1["key2"]... will be created
```

### Count

- Used as you would use it on any other component
- Create multiple instances of the same module

```
module "module1" {
 source = "./module1"
 count  = 3
}

# module.module1[0]... will be created
# module.module1[1]... will be created
# module.module1[2]... will be created
```

## 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)
- Used for multiple resources, a single resource module is usually a bad practice
- Use defaults or optionals depending on the data type of your variables
- Use dynamic blocks
- Use ternary operators and take advantage of terraform built-in functions
- Test your modules