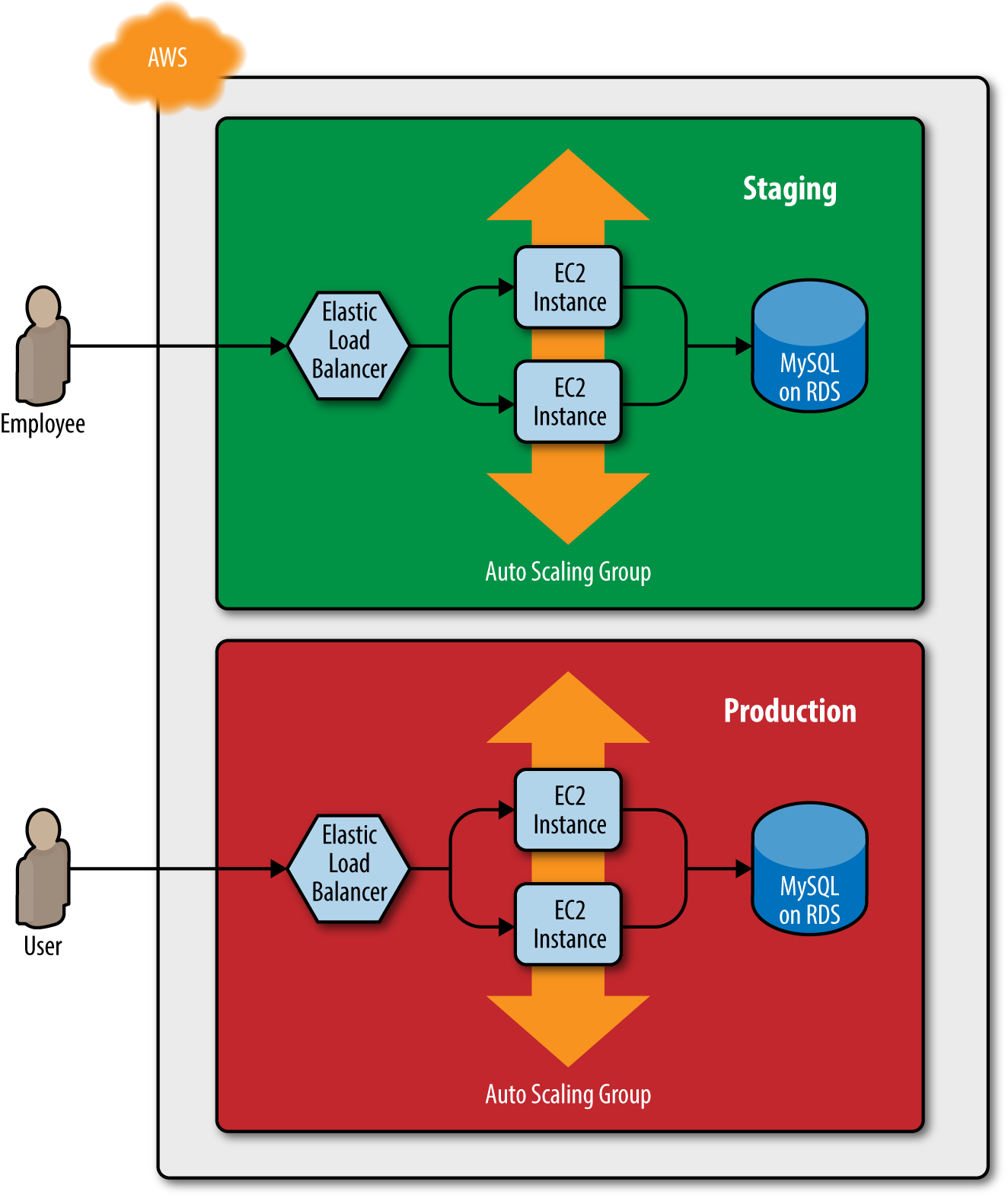
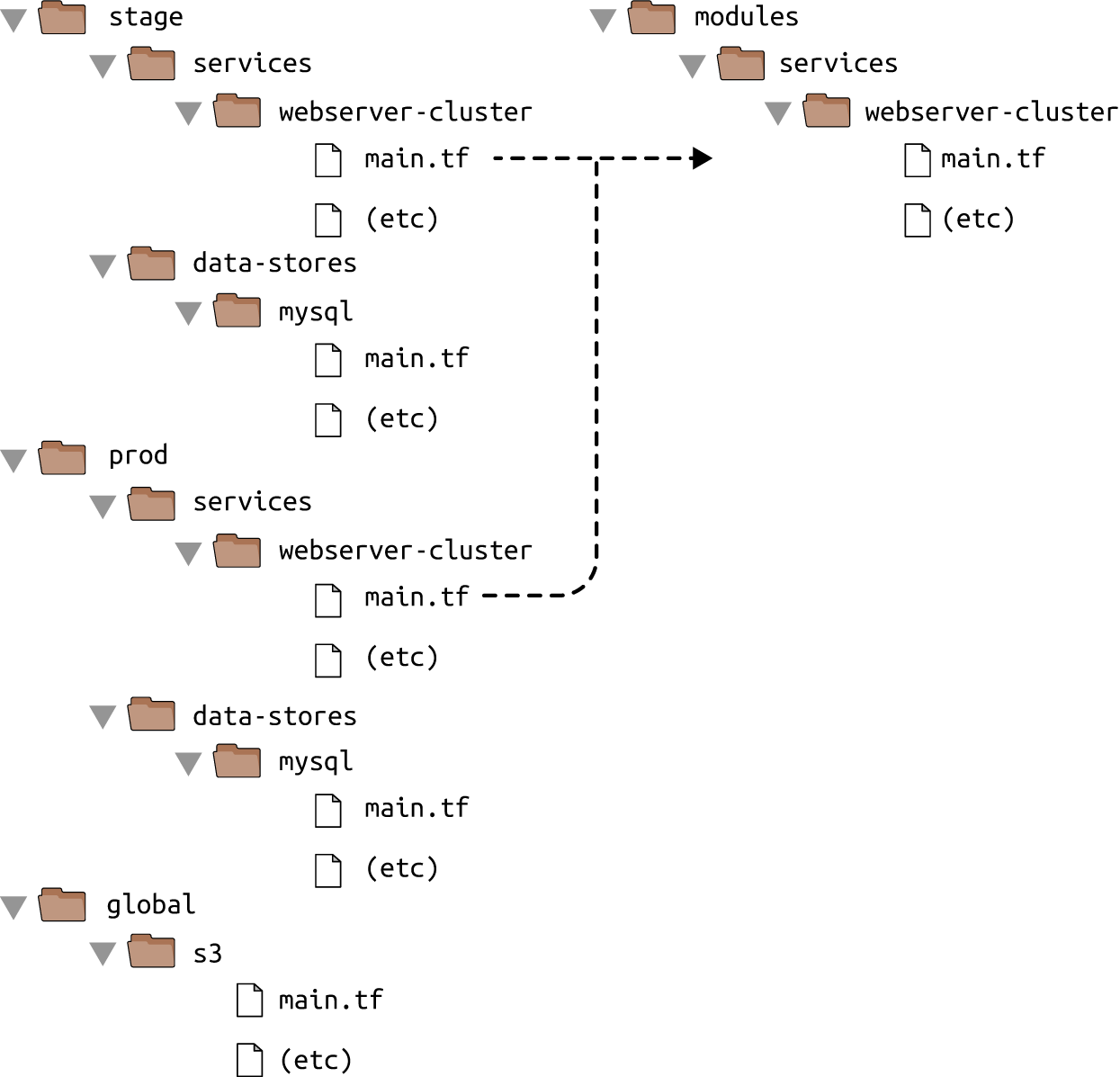
**Reusable Infrastructure with Terraform Modules**

This works great as a first environment, but you typically need at least two environments: one for your team’s internaltesting (“staging”) and one that real users can access (“production”),



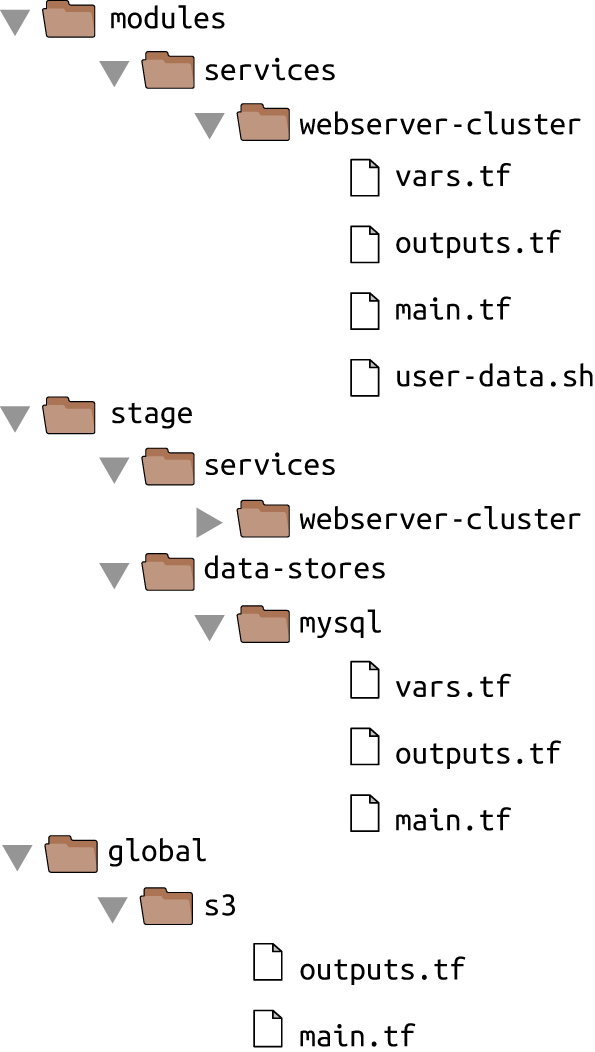
How do you add this production environment without having to copy and paste all of the code from staging?

With Terraform, you can put your code insideof a Terraform module and reuse that module in multiple places throughout your code. Instead of having the same codecopied and pasted in the staging and production environments, you’ll be able to have both environments reuse code from thesame module, as shown



Modules are the key ingredient to writing reusable, maintainable, and testable Terraform code.

like below we can use modules and structure will be like below:



You can now make use of this module in the staging environment. Here’s the syntax for using a module:

**module** "<NAME>" {

source = "<SOURCE>"

[CONFIG ...]

}

where NAME is an identifier you can use throughout the Terraform code to refer to this module (e.g., web-service),SOURCE is the path where the module code can be found

**provider "aws" {**

**region = "us-east-2"**

**}**

**module "webserver\_cluster" {**

**source = "../../../modules/services/webserver-cluster"**

**}**

## module locals

<https://www.terraform.io/docs/configuration/locals.html>

Using input variables to define your module’s inputs is great, but what if you need a way to define a variable in yourmodule to do some intermediary calculation, or just to keep your code DRY, but you don’t want to expose that variableas a configurable input? For example, the load balancer in the webserver-cluster module inmodules/services/webserver-cluster/main.tf listens on port 80, the default port for HTTP. This port number iscurrently copied and pasted in multiple places, including the load balancer listener:

you can define these as local values in a locals block:

**locals {**

**service\_name = "forum"**

**owner = "Community Team"**

**}**

Local values allow you to assign a name to any Terraform expression, and to use that name throughout the module. Thesenames are only visible within the module, so they will have no impact on other modules, and you can’t override thesevalues from outside of the module. To read the value of a local, you need to use a local reference, which uses thefollowing syntax:

**local.<NAME>**

**load\_balancer\_arn = aws\_lb.example.arn**

**port = local.http\_port**

**protocol = "HTTP"**

## Module Outputs

<https://www.terraform.io/docs/configuration/outputs.html>

In Terraform, a module can also return values. Again, you do this using a mechanism you already know: output variables. You can add the ASG name as an output variable in */modules/mod-name /outputs.tf* as follows:

output "asg\_name" {

value = aws\_autoscaling\_group.example.name

description = "The name of the Auto Scaling Group"

}

You can access module output variables using the following syntax:

**module.<MODULE\_NAME>.<OUTPUT\_NAME>**

usage as

**resource** "aws\_autoscaling\_schedule" "scale\_in\_at\_night" {

scheduled\_action\_name = "scale-in-at-night"

min\_size = 2

max\_size = 10

desired\_capacity = 2

recurrence = "0 17 \* \* \*"

autoscaling\_group\_name = **module**.webserver\_cluster.asg\_name

}

## Module Gotchas

### File paths

*By default, Terraform interprets the path relative to the current working directory. That works if you’re using the file function in a Terraform configuration file that’s in the same directory as where you’re running terraform apply (that is, if you’re using the file function in the root module), but that won’t work when you’re using file in a module that’s defined in a separate folder.*

*To solve this issue, you can use an expression known as a path reference, which is of the form****path.<TYPE>.*** *Terraform supports the following types of path references:*

***path.module***

*Returns the filesystem path of the module where the expression is defined.*

***path.root***

*Returns the filesystem path of the root module.*

***path.cwd***

*Returns the filesystem path of the current working directory. In normal use of Terraform this is the same as path.root, but some advanced uses of Terraform run it from a directory other than the root module directory, causing these paths to be different.*

*For the User Data script, you need a path relative to the module itself, so you should use path.module in the template\_file data source in*modules/services/webserver-cluster/main.tf*:*

data "template\_file" "user\_data" {

template = file("${path.module}/user-data.sh")

vars = {

server\_port = var.server\_port

db\_address = data.terraform\_remote\_state.db.outputs.address

db\_port = data.terraform\_remote\_state.db.outputs.port

}

}

## Inline Blocks

*The configuration for some Terraform resources can be defined either as inline blocks or as separate resources. When creating a module, you should always prefer using a separate resource.*

*For example, the aws\_security\_group resource allows you to define ingress and egress rules via inline blocks, as you saw in the webserver-cluster module (*modules/services/webserver-cluster/main.tf*):*

**resource** "aws\_security\_group" "alb" {

name = "${var.cluster\_name}-alb"

**ingress** {

from\_port = local.http\_port

to\_port = local.http\_port

protocol = local.tcp\_protocol

cidr\_blocks = local.all\_ips

}

**egress** {

from\_port = local.any\_port

to\_port = local.any\_port

protocol = local.any\_protocol

cidr\_blocks = local.all\_ips

}

}

*You should change this module to define the exact same ingress and egress rules by using separate aws\_security\_group\_rule resources (make sure to do this for both security groups in the module):*

**resource** "aws\_security\_group" "alb" {

name = "${var.cluster\_name}-alb"

}

**resource** "aws\_security\_group\_rule" "allow\_http\_inbound" {

type = "ingress"

security\_group\_id = aws\_security\_group.alb.id

from\_port = local.http\_port

to\_port = local.http\_port

protocol = local.tcp\_protocol

cidr\_blocks = local.all\_ips

}

**resource** "aws\_security\_group\_rule" "allow\_all\_outbound" {

type = "egress"

security\_group\_id = aws\_security\_group.alb.id

from\_port = local.any\_port

to\_port = local.any\_port

protocol = local.any\_protocol

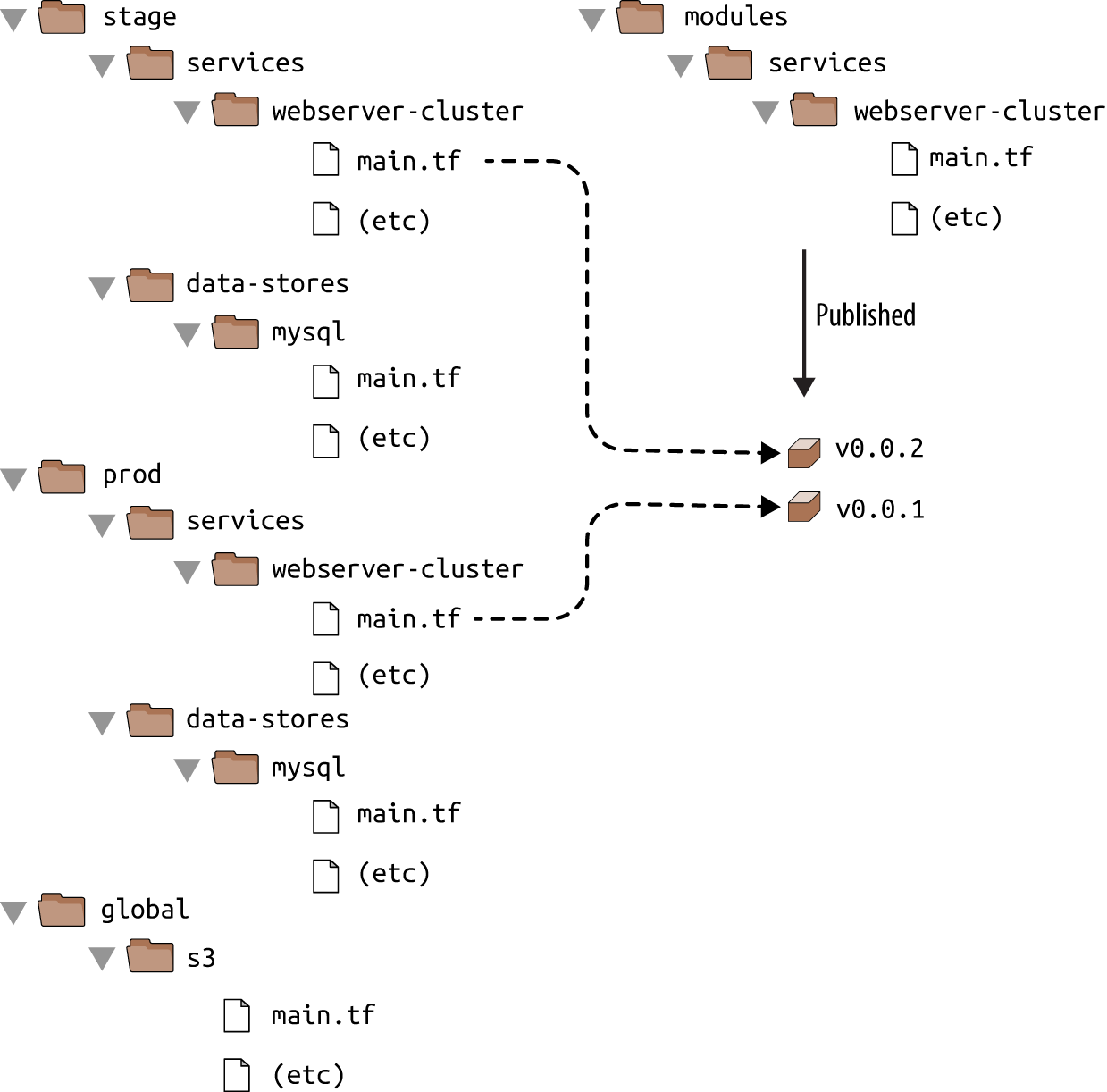
cidr\_blocks = local.all\_ips

}

*If you try to use a mix of*both*inline blocks and separate resources, you will get errors where routing rules conflict and overwrite one another. Therefore, you must use one or the other. Because of this limitation, when creating a module, you should always try to use a separate resource instead of the inline block. Otherwise, your module will be less flexible and configurable.*

## Module Versioning

If both your staging and production environment are pointing to the same module folder, as soon as you make achange in that folder, it will affect both environments on the very next deployment. This sort of coupling makes itmore difficult to test a change in staging without any chance of affecting production. A better approach is to create versionedmodules so that you can use one version in staging (e.g., v0.0.2) and a different version in production (e.g., v0.0.1),



In all of the module examples you’ve seen so far, whenever you used a module, you set the source parameter of the module to a local file path. In addition to file paths, Terraform supports other types of module sources, such as Git URLs, Mercurial URLs, and arbitrary HTTP URLs. The easiest way to create a versioned module is to put the code for the module in a separate Git repository and to set the source parameter to that repository’s URL. **That means your Terraform code will be spread out across (at least) two repositories:**

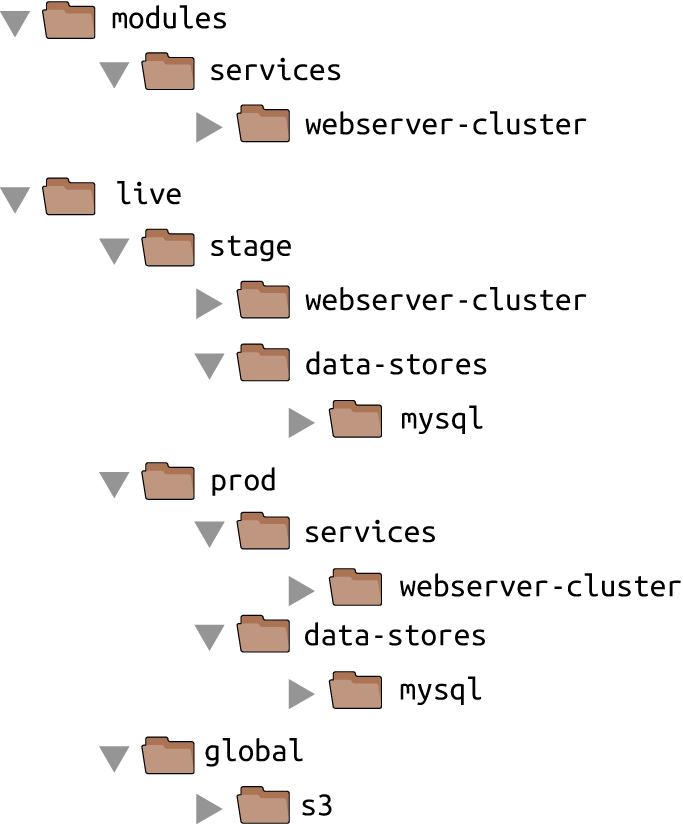
**modules**

This repo defines reusable modules. Think of each module as a “blueprint” that defines a specific part of your infrastructure.

**live**

This repo defines the live infrastructure you’re running in each environment (stage, prod, mgmt, etc.). Think of this as the “houses” you built from the “blueprints” in the modules repo.

The updated folder structure for your Terraform code nows look something like



To set up this folder structure, you’ll first need to move the stage, prod, and global folders into a folder called live. Next, configure the live and modules folders as separate Git repositories. Here is an example of how to do that for the modules folder:

$ cd modules

$ git init

$ git add .

$ git commit -m "Initial commit of modules repo"

$ git remote add origin "(URL OF REMOTE GIT REPOSITORY)"

$ git push origin master

You can also add a tag to the modules repo to use as a version number. If you’re using GitHub, you can

use the GitHub UI to [create a release](http://bit.ly/2Yv8kPg), which will create a tag under the hood. If you’re not using GitHub,

you can use the Git CLI:

$ git tag -a "v0.0.1" -m "First release of webserver-cluster module"

$ git push --follow-tags

Now you can use this versioned module in both staging and production by specifying a Git URL in the source parameter. Here is what that would look like in *live/stage/services/webserver-cluster/main.tf* if your *modules* repo was in the GitHub repo *github.com/foo/modules* (note that the double-slash in the following Git URL is required):

**module** "webserver\_cluster" {

source = "github.com/foo/modules//webserver-cluster?ref=v0.0.1"

cluster\_name = "webservers-stage"

db\_remote\_state\_bucket = "(YOUR\_BUCKET\_NAME)"

db\_remote\_state\_key = "stage/data-stores/mysql/terraform.tfstate"

instance\_type = "t2.micro"

min\_size = 2

max\_size = 2

}

If you want to try out versioned modules without messing with Git repos, you can use a module from the [code examples GitHub repo](https://github.com/brikis98/terraform-up-and-running-code) for this book (I had to break up the URL to make it fit in the book, but it should all be on one line):

source = "github.com/brikis98/terraform-up-and-running-code//

code/terraform/04-terraform-module/module-example/modules/

services/webserver-cluster?ref=v0.1.0"

The ref parameter allows you to specify a particular Git commit via its sha1 hash, a branch name, or, as in this example, a specific Git tag. I generally recommend using Git tags as version numbers for modules. Branch names are not stable, as you always get the latest commit on a branch, which may change every time you run the init command, and the sha1 hashes are not very human friendly. Git tags are as stable as a commit (in fact, a tag is just a pointer to a commit), but they allow you to use a friendly, readable name.