

## NETWORK ISOLATION

The examples in this chapter create two environments that are isolated in your Terraform code, as well as isolated in terms of having separate load balancers, servers, and databases, but they are not isolated at the network level. To keep all the examples in this book simple, all of the resources deploy into the same VPC. This means that a server in the staging environment can communicate with a server in the production environment, and vice versa.

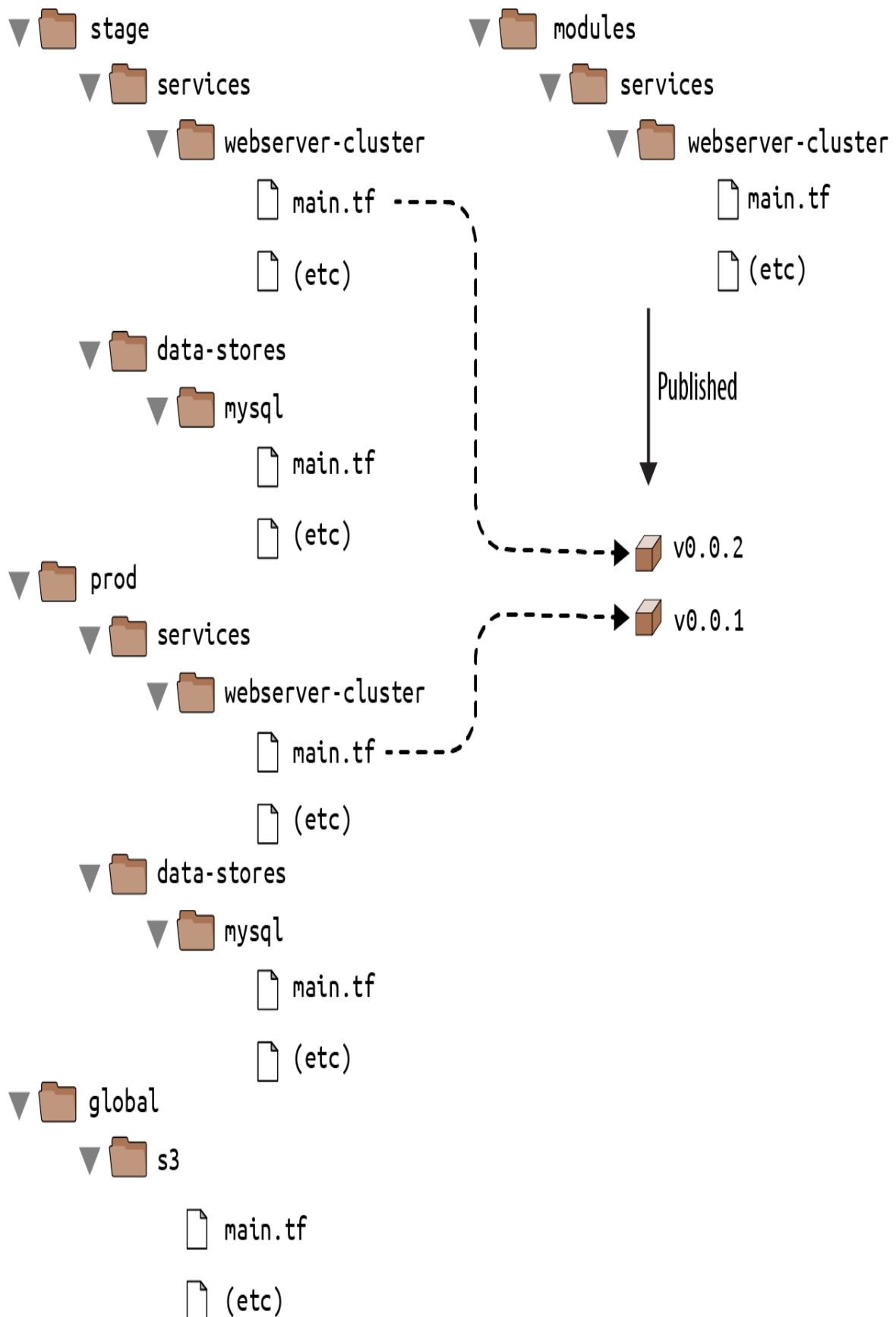
In real-world usage, running both environments in one VPC opens you up to two risks. First, a mistake in one environment could affect the other. For example, if you're making changes in staging and accidentally mess up the configuration of the route tables, all the routing in production can be affected, too. Second, if an attacker gains access to one environment, they also have access to the other. If you're making rapid changes in staging and accidentally leave a port exposed, any hacker that broke in would have access to not only your staging data but also your production data.

Therefore, outside of simple examples and experiments, you should run each environment in a separate VPC. In fact, to be extra sure, you might even run each environment in a totally separate AWS account.

## Module Versioning

If both your staging and production environment are pointing to the same module folder, as soon as you make a change in that folder, it will affect both environments on the very next deployment. This sort of coupling makes it more difficult to test a change in staging without any chance of affecting production. A better approach is to create *versioned modules* 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), as shown in [Figure 4-5](#).

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 filepath. In addition to file paths, Terraform supports other types of module sources, such as Git URLs, Mercurial URLs, and arbitrary HTTP URLs.<sup>1</sup>



*Figure 4-5. By versioning your modules, you can use different versions in different environments:  
e.g., v0.0.1 in prod and v0.0.2 in stage.*

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 now looks something like **Figure 4-6**.

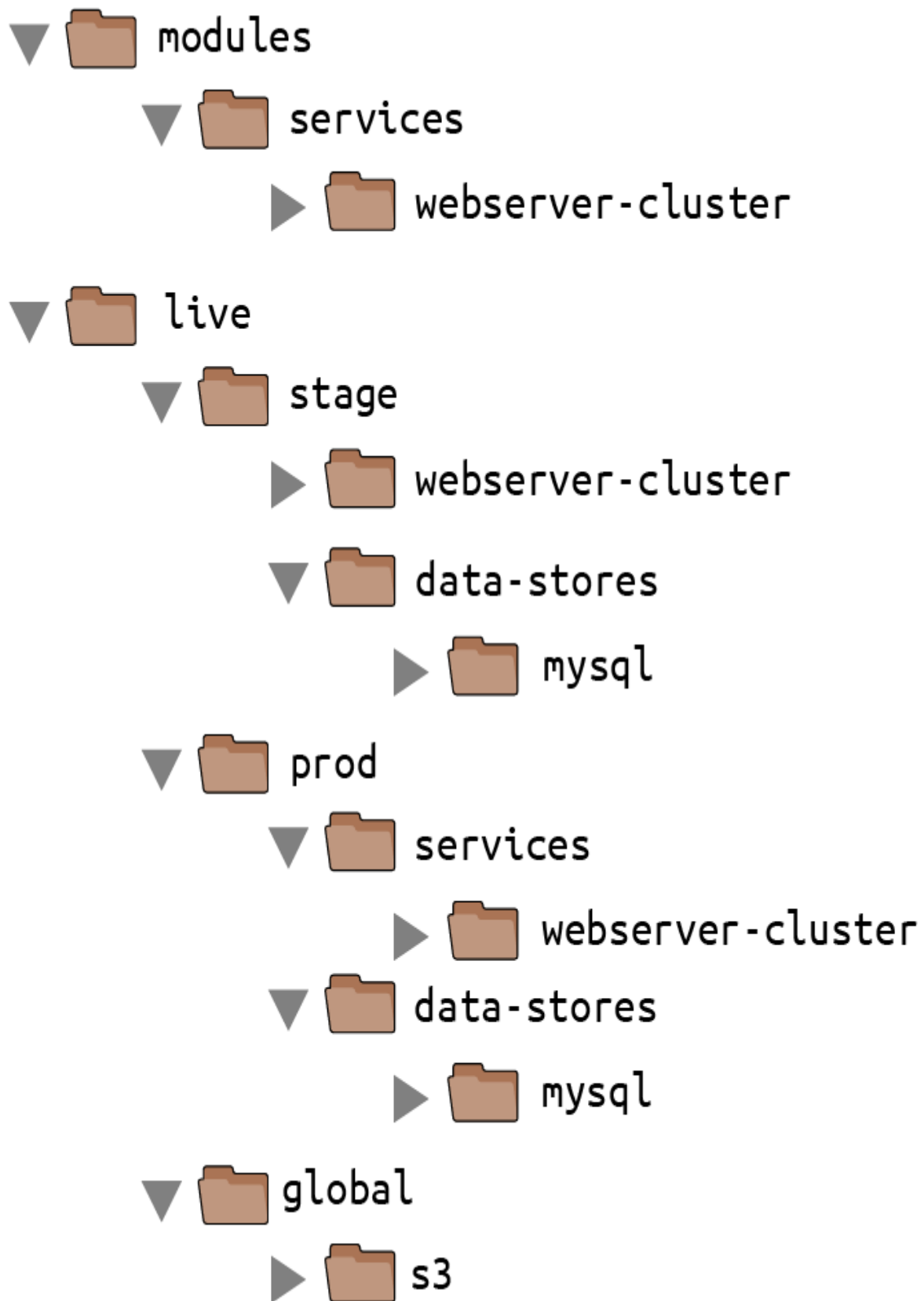


Figure 4-6. You should store reusable, versioned modules in one repo (*modules*) and the configuration for your live environments in another repo (*live*).

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 main
```

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**, 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//services/webserver-cluster?
ref=v0.0.1"

  cluster_name           = "webserver-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](#) 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=v0.3.0"
```

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.

A particularly useful naming scheme for tags is *semantic versioning*. This is a versioning scheme of the format `MAJOR.MINOR.PATCH` (e.g., `1.0.4`) with specific rules on when you should increment each part of the version number. In particular, you should increment the following:

- The `MAJOR` version when you make incompatible API changes
- The `MINOR` version when you add functionality in a backward-compatible manner
- The `PATCH` version when you make backward-compatible bug fixes

Semantic versioning gives you a way to communicate to users of your module what kinds of changes you've made and the implications of upgrading.

Because you've updated your Terraform code to use a versioned module URL, you need to instruct Terraform to download the module code by rerunning `terraform init`:

```
$ terraform init
Initializing modules...
Downloading git@github.com:brikis98/terraform-up-and-running-
code.git?ref=v0.3.0
for webserver_cluster...

(...)
```

This time, you can see that Terraform downloads the module code from Git rather than your local filesystem. After the module code has been downloaded, you can run the `apply` command as usual.

## PRIVATE GIT REPOS

If your Terraform module is in a private Git repository, to use that repo as a module source, you need to give Terraform a way to authenticate to that Git repository. I recommend using SSH auth so that you don't need to hardcode the credentials for your repo in the code itself. With SSH authentication, each developer can create an SSH key, associate it with their Git user, add it to `ssh-agent`, and Terraform will automatically use that key for authentication if you use an SSH source URL.<sup>2</sup>

The source URL should be of the form:

```
git@github.com:<OWNER>/<REPO>.git//<PATH>?ref=<VERSION>
```

For example:

```
git@github.com:acme/modules.git//example?ref=v0.1.2
```

To check that you've formatted the URL correctly, try to `git clone` the base URL from your terminal:

```
$ git clone git@github.com:acme/modules.git
```

If that command succeeds, Terraform should be able to use the private repo, too.

Now that you're using versioned modules, let's walk through the process of making changes. Let's say you made some changes to the `webserver-cluster` module, and you want to test them out in staging. First, you'd commit those changes to the *modules* repo:

```
$ cd modules
$ git add .
$ git commit -m "Made some changes to webserver-cluster"
$ git push origin main
```

Next, you would create a new tag in the *modules* repo:

```
$ git tag -a "v0.0.2" -m "Second release of webserver-cluster"
$ git push --follow-tags
```

And now you can update *just* the source URL used in the staging environment (*live/stage/services/webserver-cluster/main.tf*) to use this new version:

```
module "webserver_cluster" {
  source = "github.com/foo/modules//services/webserver-cluster?
ref=v0.0.2"

  cluster_name          = "webserver-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
}
```

In production (*live/prod/services/webserver-cluster/main.tf*), you can happily continue to run v0.0.1 unchanged:

```
module "webserver_cluster" {
  source = "github.com/foo/modules//services/webserver-cluster?
ref=v0.0.1"

  cluster_name          = "webserver-prod"
```