maintenance more difficult, because you don't have a good picture of your infrastructure.

- Putting the two previous items together, the result is that workspaces can be fairly error prone. The lack of visibility makes it easy to forget what workspace you're in and accidentally deploy changes in the wrong one (e.g., accidentally running `terraform destroy` in a "production" workspace rather than a "staging" workspace), and because you must use the same authentication mechanism for all workspaces, you have no other layers of defense to protect against such errors.

Due to these drawbacks, workspaces are not a suitable mechanism for isolating one environment from another: e.g., isolating staging from production.[5] To get proper isolation between environments, instead of workspaces, you'll most likely want to use file layout, which is the topic of the next section.

Before moving on, make sure to clean up the three EC2 Instances you just deployed by running `terraform workspace select <name>` and `terraform destroy` in each of the three workspaces.

Isolation via File Layout

To achieve full isolation between environments, you need to do the following:

- Put the Terraform configuration files for each environment into a separate folder. For example, all of the configurations for the staging environment can be in a folder called *stage* and all the configurations for the production environment can be in a folder called *prod*.
- Configure a different backend for each environment, using different authentication mechanisms and access controls: e.g., each environment could live in a separate AWS account with a separate S3 bucket as a backend.

With this approach, the use of separate folders makes it much clearer which environments you're deploying to, and the use of separate state files, with separate authentication mechanisms, makes it significantly less likely that a screw-up in one environment can have any impact on another.

In fact, you might want to take the isolation concept beyond environments and down to the "component" level, where a component is a coherent set of resources that you typically deploy together. For example, after you've set up the basic network topology for your infrastructure—in AWS lingo, your Virtual Private Cloud (VPC) and all the associated subnets, routing rules, VPNs, and network ACLs—you will probably change it only once every few months, at most. On the other hand, you might deploy a new version of a web server multiple times per day. If you manage the infrastructure for both the VPC component and the web server component in the same set of Terraform configurations, you are unnecessarily putting your entire network topology at risk of breakage (e.g., from a simple typo in the code or someone accidentally running the wrong command) multiple times per day.

Therefore, I recommend using separate Terraform folders (and therefore separate state files) for each environment (staging, production, etc.) and for each component (VPC, services, databases) within that environment. To see what this looks like in practice, let's go through the recommended file layout for Terraform projects.

Figure 3-7 shows the file layout for my typical Terraform project.

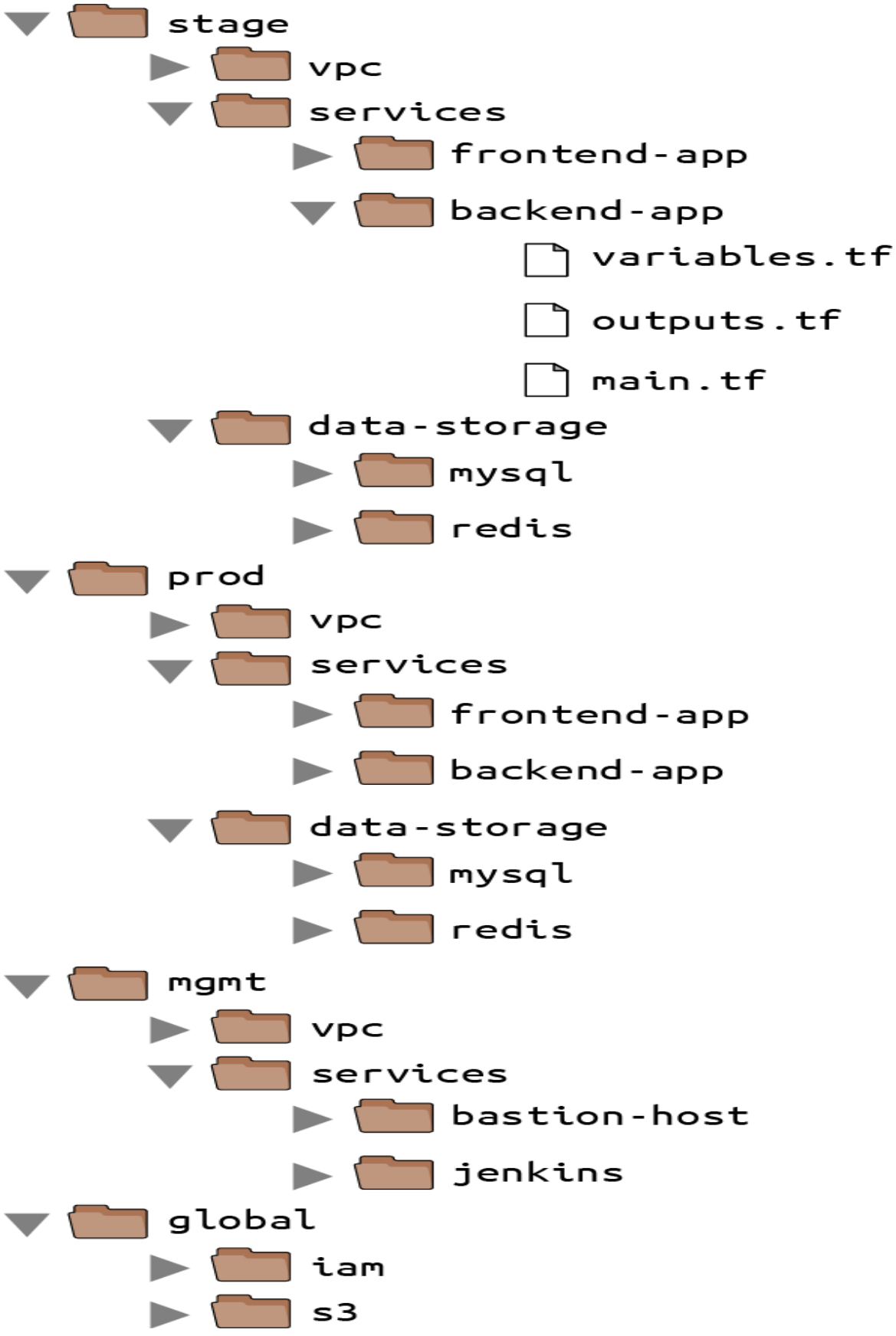
stage
vpc
services
frontend-app
backend-app
variables.tf
outputs.tf
main.tf
data-storage
mysql
redis
prod
vpc
services
frontend-app
backend-app
data-storage
mysql
redis
mgmt
vpc
services
bastion-host
jenkins
global
iam
s3

Figure 3-7. The typical file layout for a Terraform project uses separate folders for each environment and for each component within that environment.

At the top level, there are separate folders for each "environment." The exact environments differ for every project, but the typical ones are as follows:

stage

An environment for pre-production workloads (i.e., testing)

prod

An environment for production workloads (i.e., user-facing apps)

mgmt

An environment for DevOps tooling (e.g., bastion host, CI server)

global

A place to put resources that are used across all environments (e.g., S3, IAM)

Within each environment, there are separate folders for each "component." The components differ for every project, but here are the typical ones:

vpc

The network topology for this environment.

services

The apps or microservices to run in this environment, such as a Ruby on Rails frontend or a Scala backend. Each app could even live in its own folder to isolate it from all the other apps.

data-storage

The data stores to run in this environment, such as MySQL or Redis. Each data store could even reside in its own folder to isolate it from all

other data stores.

Within each component, there are the actual Terraform configuration files, which are organized according to the following naming conventions:

variables.tf

Input variables

outputs.tf

Output variables

main.tf

Resources and data sources

When you run Terraform, it simply looks for files in the current directory with the *.tf* extension, so you can use whatever filenames you want. However, although Terraform may not care about filenames, your teammates probably do. Using a consistent, predictable naming convention makes your code easier to browse: e.g., you'll always know where to look to find a variable, output, or resource.

Note that the preceding convention is the *minimum* convention you should follow, because in virtually all uses of Terraform, it's useful to be able to jump to the input variables, output variables, and resources very quickly, but you may want to go beyond this convention. Here are just a few examples:

dependencies.tf

It's common to put all your data sources in a *dependencies.tf* file to make it easier to see what external things the code depends on.

providers.tf

You may want to put your `provider` blocks into a *providers.tf* file so you can see, at a glance, what providers the code talks to and what

authentication you'll have to provide.

main-xxx.tf

If the *main.tf* file is getting really long because it contains a large number of resources, you could break it down into smaller files that group the resources in some logical way: e.g., *main-iam.tf* could contain all the IAM resources, *main-s3.tf* could contain all the S3 resources, and so on. Using the *main-* prefix makes it easier to scan the list of files in a folder when they are organized alphabetically, as all the resources will be grouped together. It's also worth noting that if you find yourself managing a very large number of resources and struggling to break them down across many files, that might be a sign that you should break your code into smaller modules instead, which is a topic I'll dive into in Chapter 4.

Let's take the web server cluster code you wrote in Chapter 2, plus the Amazon S3 and DynamoDB code you wrote in this chapter, and rearrange it using the folder structure in Figure 3-8.

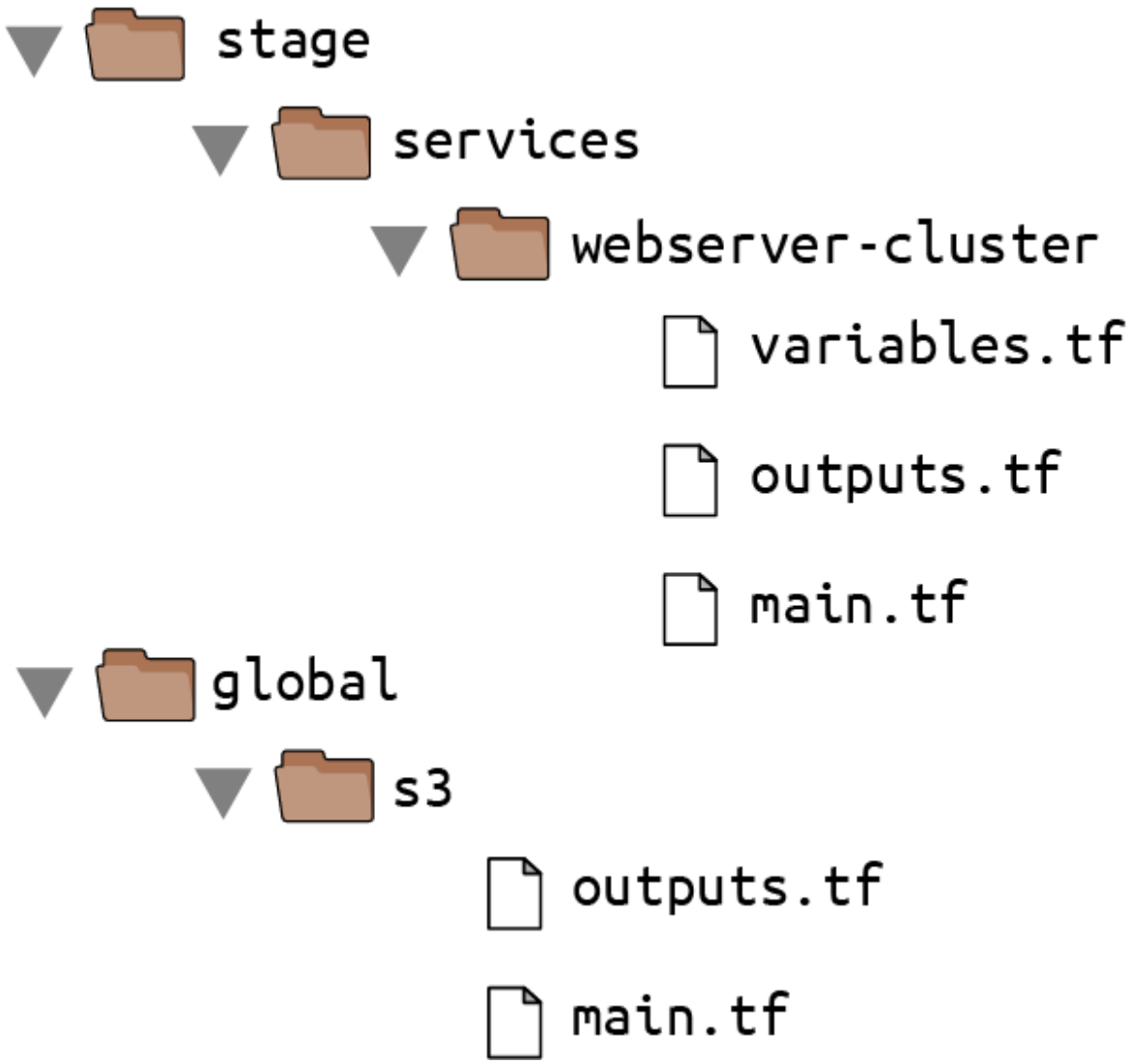

Figure 3-8. Move the web server cluster code into a stage/services/webserver-cluster folder to indicate that this is a testing or staging version of the web server.

The S3 bucket you created in this chapter should be moved into the *global/s3* folder. Move the output variables (`s3_bucket_arn` and `dynamodb_table_name`) into *outputs.tf*. When moving the folder, make sure that you don't miss the (hidden) *.terraform* folder when copying files to the new location so you don't need to reinitialize everything.

The web server cluster you created in Chapter 2 should be moved into *stage/services/webserver-cluster* (think of this as the "testing" or "staging" version of that web server cluster; you'll add a "production" version in the

next chapter). Again, make sure to copy over the *.terraform* folder, move input variables into *variables.tf*, and move output variables into *outputs.tf*.

You should also update the web server cluster to use S3 as a `backend`. You can copy and paste the `backend` config from *global/s3/main.tf* more or less verbatim, but make sure to change the `key` to the same folder path as the web server Terraform code: *stage/services/webserver-cluster/terraform.tfstate*. This gives you a 1:1 mapping between the layout of your Terraform code in version control and your Terraform state files in S3, so it's obvious how the two are connected. The `s3` module already sets the `key` using this convention.

This file layout has a number of advantages:

Clear code / environment layout

> It's easy to browse the code and understand exactly what components are deployed in each environment.

Isolation

> This layout provides a good amount of isolation between environments and between components within an environment, ensuring that if something goes wrong, the damage is contained as much as possible to just one small part of your entire infrastructure.

In some ways, these advantages are drawbacks, too:

Working with multiple folders

> Splitting components into separate folders prevents you from accidentally blowing up your entire infrastructure in one command, but it also prevents you from creating your entire infrastructure in one command. If all of the components for a single environment were defined in a single Terraform configuration, you could spin up an entire environment with a single call to `terraform apply`. But if all of the components are in separate folders, then you need to run `terraform apply` separately in each one.