Chapter 4. How to Create Reusable Infrastructure with Terraform Modules

At the end of Chapter 3, you deployed the architecture shown in Figure 4-1.

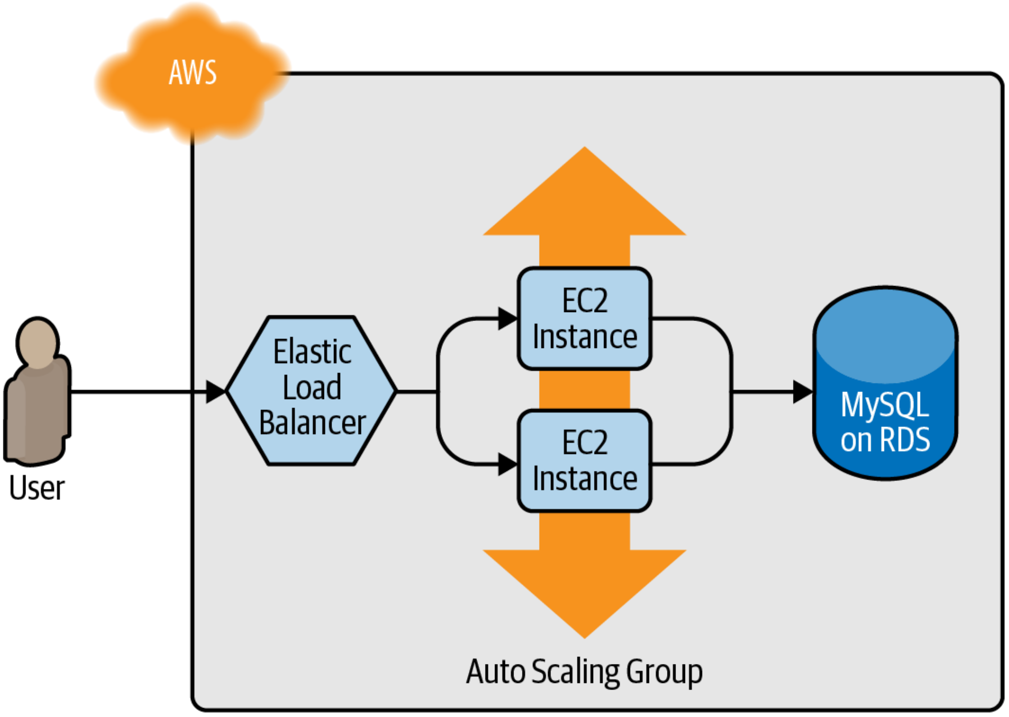

Figure 4-1. The architecture you deployed in previous chapters included a load balancer, web server cluster, and database.

This works great as a first environment, but you typically need at least two environments: one for your team's internal testing ("staging") and one that real users can access ("production"), as shown in Figure 4-2. Ideally, the two environments are nearly identical, though you might run slightly fewer/smaller servers in staging to save money.

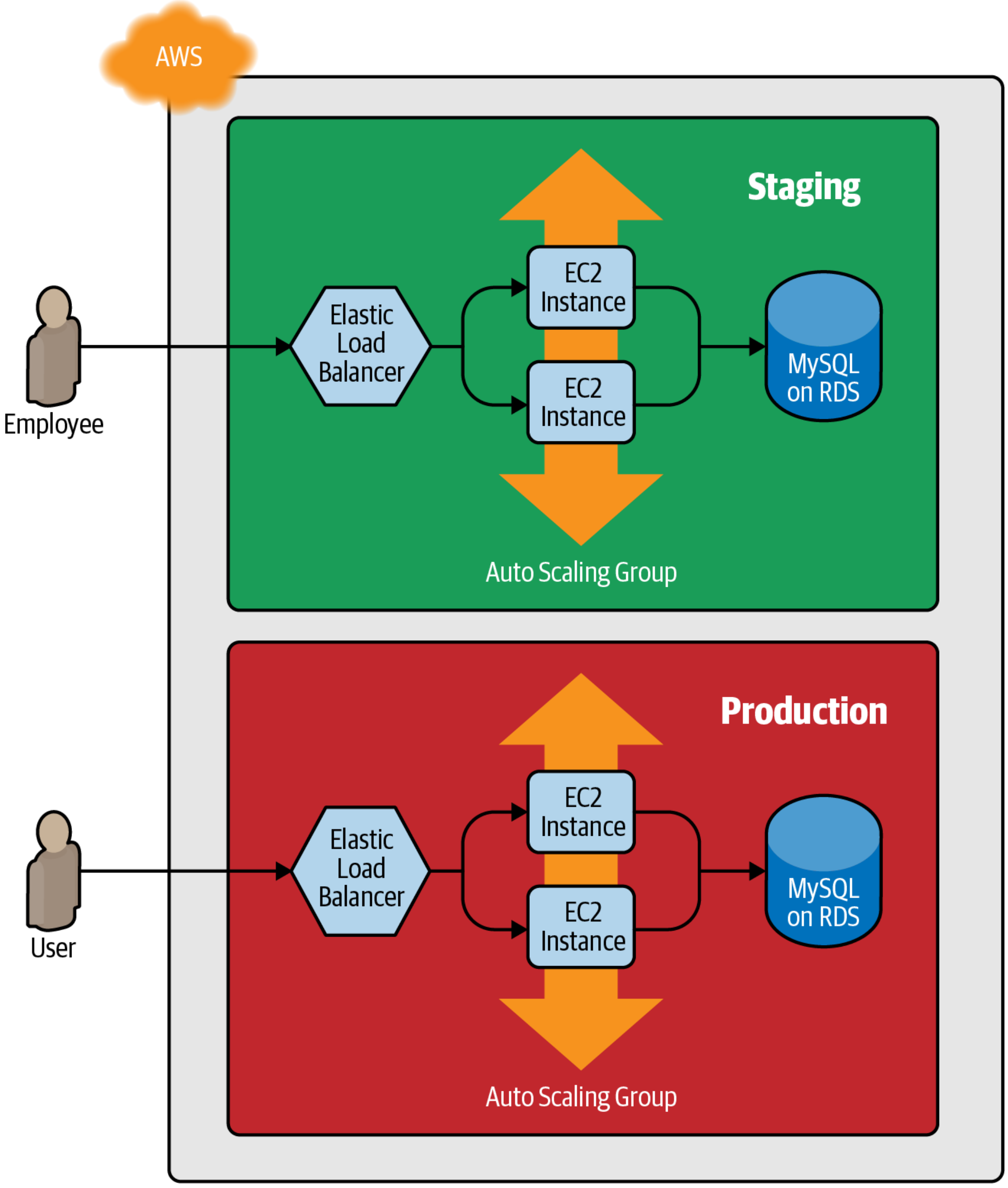

Figure 4-2. The architecture you'll deploy in this chapter will have two environments, each with its own load balancer, web server cluster, and database.

How do you add this production environment without having to copy and paste all of the code from staging? For example, how do you avoid having to copy and paste all the code in *stage/services/webserver-cluster* into *prod/services/webserver-cluster* and all the code in *stage/data-stores/mysql* into *prod/data-stores/mysql*?

In a general-purpose programming language such as Ruby, if you had the same code copied and pasted in several places, you could put that code inside of a function and reuse that function everywhere:

```
# Define the function in one place
def example_function()
  puts "Hello, World"
end

# Use the function in multiple other places
example_function()
```

With Terraform, you can put your code inside of a *Terraform module* and reuse that module in multiple places throughout your code. Instead of having the same code copied and pasted in the staging and production environments, you'll be able to have both environments reuse code from the same module, as shown in Figure 4-3.

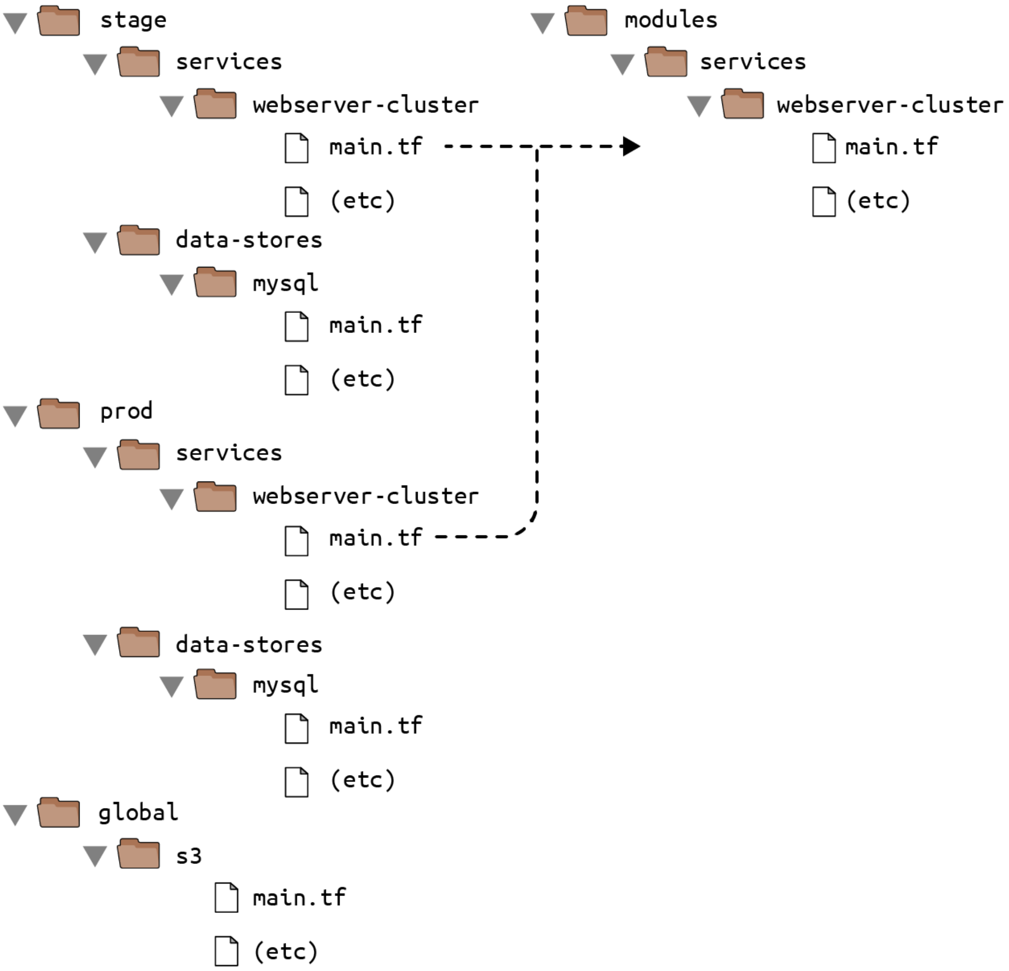

Figure 4-3. Putting code into modules allows you to reuse that code from multiple environments.

This is a big deal. Modules are the key ingredient to writing reusable, maintainable, and testable Terraform code. Once you start using them, there's no going back. You'll start building everything as a module, creating a library of modules to share within your company, using modules that you find online, and thinking of your entire infrastructure as a collection of reusable modules.

In this chapter, I'll show you how to create and use Terraform modules by covering the following topics:

- Module basics
- Module inputs
- Module locals
- Module outputs
- Module gotchas
- Module versioning

EXAMPLE CODE

As a reminder, you can find all of the code examples in the book on GitHub.

Module Basics

A Terraform module is very simple: any set of Terraform configuration files in a folder is a module. All of the configurations you've written so far have technically been modules, although not particularly interesting ones, since you deployed them directly: if you run `apply` directly on a module, it's referred to as a *root module*. To see what modules are really capable of, you need to create a *reusable module*, which is a module that is meant to be used within other modules.

As an example, let's turn the code in *stage/services/webserver-cluster*, which includes an Auto Scaling Group (ASG), Application Load Balancer (ALB), security groups, and many other resources, into a reusable module.

As a first step, run `terraform destroy` in the *stage/services/webserver-cluster* to clean up any resources that you created earlier. Next, create a new top-level folder called *modules*, and move all of the files from *stage/services/webserver-cluster* to *modules/services/webserver-cluster*. You should end up with a folder structure that looks something like Figure 4-4.

Open up the *main.tf* file in *modules/services/webserver-cluster*, and remove the `provider` definition. Providers should be configured only in root modules and not in reusable modules (you'll learn a lot more about working with providers in Chapter 7).

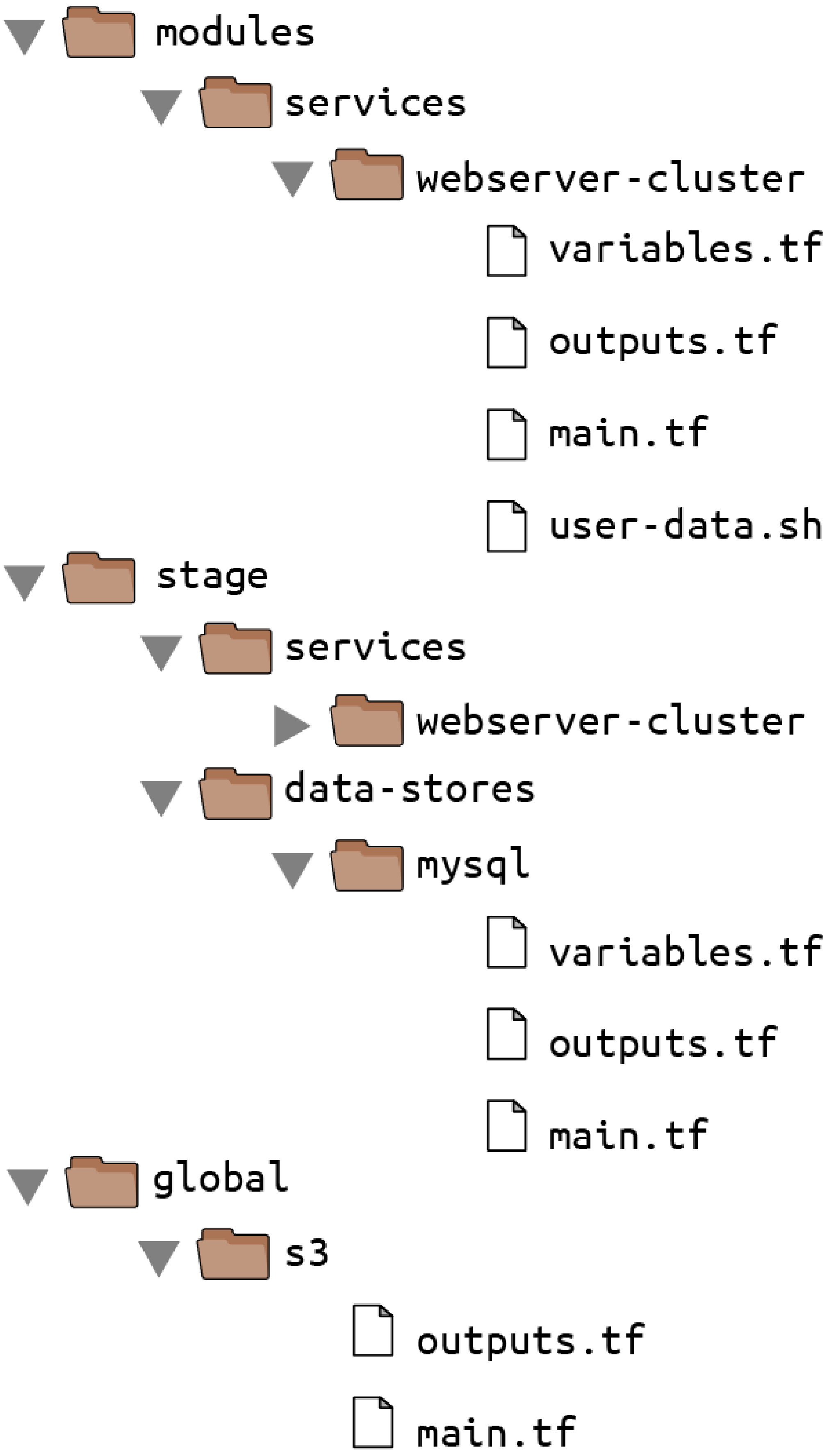

Figure 4-4. Move your reusable web server cluster code into a modules/services/web server-cluster folder.

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., webserver_cluster), SOURCE is the path where the module code can be found (e.g., *modules/services/webserver-cluster*), and CONFIG consists of arguments that are specific to that module. For example, you can create a new file in *stage/services/webserver-cluster/main.tf* and use the webserver-cluster module in it as follows:

```
provider "aws" {
  region = "us-east-2"
}

module "webserver_cluster" {
  source = "../../../modules/services/webserver-cluster"
}
```

You can then reuse the exact same module in the production environment by creating a new *prod/services/webserver-cluster/main.tf* file with the following contents:

```
provider "aws" {
  region = "us-east-2"
}

module "webserver_cluster" {
  source = "../../../modules/services/webserver-cluster"
}
```

And there you have it: code reuse in multiple environments that involves minimal duplication. Note that whenever you add a module to your Terraform configurations or modify the source parameter of a module, you need to run the init command before you run plan or apply:

```
$ terraform init
Initializing modules...
- webserver_cluster in ../../../modules/services/webserver-cluster

Initializing the backend...

Initializing provider plugins...

Terraform has been successfully initialized!
```

Now you've seen all the tricks the `init` command has up its sleeve: it installs providers, it configures your backends, and it downloads modules, all in one handy command.

Before you run the `apply` command on this code, be aware that there is a problem with the `webserver-cluster` module: all of the names are hardcoded. That is, the name of the security groups, ALB, and other resources are all hardcoded, so if you use this module more than once in the same AWS account, you'll get name conflict errors. Even the details for how to read the database's state are hardcoded because the *main.tf* file you copied into *modules/services/webserver-cluster* is using a `terraform_remote_state` data source to figure out the database address and port, and that `terraform_remote_state` is hardcoded to look at the staging environment.

To fix these issues, you need to add configurable inputs to the `webserver-cluster` module so that it can behave differently in different environments.

Module Inputs

To make a function configurable in a general-purpose programming language such as Ruby, you can add input parameters to that function:

```
# A function with two input parameters
def example_function(param1, param2)
  puts "Hello, #{param1} #{param2}"
end

# Pass two input parameters to the function
example_function("foo", "bar")
```

In Terraform, modules can have input parameters, too. To define them, you use a mechanism you're already familiar with: input variables. Open up *modules/services/webserver-cluster/variables.tf* and add three new input variables:

```
variable "cluster_name" {
  description = "The name to use for all the cluster resources"
  type        = string
}

variable "db_remote_state_bucket" {
  description = "The name of the S3 bucket for the database's remote state"
  type        = string
}

variable "db_remote_state_key" {
  description = "The path for the database's remote state in S3"
  type        = string
}
```

Next, go through *modules/services/webserver-cluster/main.tf*, and use `var.cluster_name` instead of the hardcoded names (e.g., instead of `"terraform-asg-example"`). For example, here is how you do it for the ALB security group:

```
resource "aws_security_group" "alb" {
  name = "${var.cluster_name}-alb"

  ingress {
    from_port   = 80
    to_port     = 80
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

Notice how the `name` parameter is set to `"${var.cluster_name}-alb"`. You'll need to make a similar change to the other `aws_security_group` resource (e.g., give it the name `"${var.cluster_name}-instance"`), the `aws_alb` resource, and the `tag` section of the `aws_autoscaling_group` resource.

You should also update the `terraform_remote_state` data source to use the `db_remote_state_bucket` and `db_remote_state_key` as its `bucket` and `key` parameter, respectively, to ensure you're reading the state file from the right environment:

```
data "terraform_remote_state" "db" {
  backend = "s3"

  config = {
    bucket = var.db_remote_state_bucket
    key    = var.db_remote_state_key
    region = "us-east-2"
    }
  }
```

Now, in the staging environment, in *stage/services/webserver-cluster/main.tf*, you can set these new input variables accordingly:

```
module "webserver_cluster" {
  source = "../../../modules/services/webserver-cluster"

  cluster_name           = "webservers-stage"
  db_remote_state_bucket = "(YOUR_BUCKET_NAME)"
  db_remote_state_key    = "stage/data-stores/mysql/terraform.tfstate"
}
```

You should also set these variables in the production environment in *prod/services/webserver-cluster/main.tf* but to different values that correspond to that environment:

```
module "webserver_cluster" {
  source = "../../../modules/services/webserver-cluster"

  cluster_name           = "webservers-prod"
  db_remote_state_bucket = "(YOUR_BUCKET_NAME)"
  db_remote_state_key    = "prod/data-stores/mysql/terraform.tfstate"
}
```

NOTE

The production database doesn't actually exist yet. As an exercise, I leave it up to you to add a production database similar to the staging one.

As you can see, you set input variables for a module by using the same syntax as setting arguments for a resource. The input variables are the API of the module, controlling how it will behave in different environments.

So far, you've added input variables for the name and database remote state, but you may want to make other parameters configurable in your module, too. For example, in staging, you might want to run a small web server cluster to save money, but in production, you might want to run a larger cluster to handle lots of traffic. To do that, you can add three more input variables to *modules/services/webserver-cluster/variables.tf*:

```
variable "instance_type" {
  description = "The type of EC2 Instances to run (e.g. t2.micro)"
  type        = string
}

variable "min_size" {
  description = "The minimum number of EC2 Instances in the ASG"
  type        = number
}

variable "max_size" {
  description = "The maximum number of EC2 Instances in the ASG"
  type        = number
}
```

Next, update the launch configuration in *modules/services/webserver-cluster/main.tf* to set its `instance_type` parameter to the new `var.instance_type` input variable:

```
resource "aws_launch_configuration" "example" {
  image_id        = "ami-0fb653ca2d3203ac1"
  instance_type   = var.instance_type
  security_groups = [aws_security_group.instance.id]

  user_data = templatefile("user-data.sh", {
    server_port = var.server_port
    db_address  = data.terraform_remote_state.db.outputs.address
    db_port     = data.terraform_remote_state.db.outputs.port
  })

  # Required when using a launch configuration with an auto scaling group.
  lifecycle {
    create_before_destroy = true
  }
}
```

Similarly, you should update the ASG definition in the same file to set its `min_size` and `max_size` parameters to the new `var.min_size` and `var.max_size` input variables, respectively:

```
resource "aws_autoscaling_group" "example" {
  launch_configuration = aws_launch_configuration.example.name
  vpc_zone_identifier  = data.aws_subnets.default.ids
  target_group_arns    = [aws_lb_target_group.asg.arn]
  health_check_type    = "ELB"

  min_size = var.min_size
  max_size = var.max_size
  tag {
    key                 = "Name"
    value               = var.cluster_name
    propagate_at_launch = true
  }
}
```

Now, in the staging environment (*stage/services/webserver-cluster/main.tf*), you can keep the cluster small and inexpensive by setting `instance_type` to `"t2.micro"` and `min_size` and `max_size` to `2`:

```
module "webserver_cluster" {
  source = "../../../modules/services/webserver-cluster"

  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
}
```

On the other hand, in the production environment, you can use a larger `instance_type` with more CPU and memory, such as `m4.large` (be aware that this Instance type is *not* part of the AWS Free Tier, so if you're just using this for learning and don't want to be charged, stick with `"t2.micro"` for the `instance_type`), and you can set `max_size` to `10` to allow the cluster to shrink or grow depending on the load (don't worry, the cluster will launch with two Instances initially):

```
module "webserver_cluster" {
  source = "../../../modules/services/webserver-cluster"

  cluster_name           = "webservers-prod"
  db_remote_state_bucket = "(YOUR_BUCKET_NAME)"
  db_remote_state_key    = "prod/data-stores/mysql/terraform.tfstate"

  instance_type = "m4.large"
  min_size      = 2
  max_size      = 10
}
```

Module Locals

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

```
resource "aws_lb_listener" "http" {
  load_balancer_arn = aws_lb.example.arn
  port              = 80
  protocol          = "HTTP"

  # By default, return a simple 404 page
  default_action {
    type = "fixed-response"

    fixed_response {
      content_type = "text/plain"
      message_body = "404: page not found"
      status_code  = 404
    }
  }
}
```

And the load balancer security group:

```
resource "aws_security_group" "alb" {
  name = "${var.cluster_name}-alb"

  ingress {
    from_port   = 80
    to_port     = 80
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

The values in the security group, including the “all IPs” CIDR block `0.0.0.0/0`, the “any port” value of `0`, and the “any protocol” value of `"-1"` are also copied and pasted in several places throughout the module. Having these magical values hardcoded in multiple places makes the code more difficult to read and maintain. You could extract values into input variables, but then users of your module will be able to (accidentally) override these values, which you might not want. Instead of using input variables, you can define these as *local values* in a `locals` block:

```
locals {
  http_port    = 80
  any_port     = 0
  any_protocol = "-1"
  tcp_protocol = "tcp"
  all_ips      = ["0.0.0.0/0"]
}
```

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

```
local.<NAME>
```

Use this syntax to update the `port` parameter of your load-balancer listener:

```
resource "aws_lb_listener" "http" {
  load_balancer_arn = aws_lb.example.arn
  port              = local.http_port
  protocol          = "HTTP"

  # By default, return a simple 404 page
  default_action {
    type = "fixed-response"

    fixed_response {
      content_type = "text/plain"
      message_body = "404: page not found"
      status_code  = 404
    }
  }
}
```

Similarly, update virtually all the parameters in the security groups in the module, including the load-balancer security group:

```
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
  }
}
```

Locals make your code easier to read and maintain, so use them often.

Module Outputs

A powerful feature of ASGs is that you can configure them to increase or decrease the number of servers you have running in response to load. One way to do this is to use a *scheduled action*, which can change the size of the cluster at a scheduled time during the day. For example, if traffic to your cluster is much higher during normal business hours, you can use a scheduled action to increase the number of servers at 9 a.m. and decrease it at 5 p.m.

If you define the scheduled action in the `webserver-cluster` module, it would apply to both staging and production. Because you don't need to do this sort of scaling in your staging environment, for the time being, you can define the auto scaling schedule directly in the production configurations (in Chapter 5, you'll see how to conditionally define resources, which lets you move the scheduled action into the `webserver-cluster` module).

To define a scheduled action, add the following two `aws_autoscaling_schedule` resources to *prod/services/webserver-cluster/main.tf*:

```
resource "aws_autoscaling_schedule" "scale_out_during_business_hours" {
  scheduled_action_name = "scale-out-during-business-hours"
  min_size              = 2
  max_size              = 10
  desired_capacity      = 10
  recurrence            = "0 9 * * *"
}

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 * * *"
}
```

This code uses one `aws_autoscaling_schedule` resource to increase the number of servers to 10 during the morning hours (the `recurrence` parameter uses cron syntax, so `"0 9 * * *"` means “9 a.m. every day”) and a second `aws_autoscaling_schedule` resource to decrease the number of servers at night (`"0 17 * * *"` means “5 p.m. every day”). However, both usages of `aws_autoscaling_schedule` are missing a required parameter, `autoscaling_group_name`, which specifies the name of the ASG. The ASG itself is defined within the `webserver-cluster` module, so how do you access its name? In a general-purpose programming language such as Ruby, functions can return values:

```
# A function that returns a value
def example_function(param1, param2)
  return "Hello, #{param1} #{param2}"
end

# Call the function and get the return value
return_value = example_function("foo", "bar")
```

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/services/webserver-cluster/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>
```

For example:

```
module.frontend.asg_name
```

In *prod/services/webserver-cluster/main.tf*, you can use this syntax to set the `autoscaling_group_name` parameter in each of the `aws_autoscaling_schedule` resources:

```
resource "aws_autoscaling_schedule" "scale_out_during_business_hours" {
  scheduled_action_name = "scale-out-during-business-hours"
  min_size              = 2
  max_size              = 10
  desired_capacity      = 10
  recurrence            = "0 9 * * *"

  autoscaling_group_name = module.webserver_cluster.asg_name
}

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
}
```

You might want to expose one other output in the `webserver-cluster` module: the DNS name of the ALB, so you know what URL to test when the cluster is deployed. To do that, you again add an output variable in */modules/services/webserver-cluster/outputs.tf*:

```
output "alb_dns_name" {
  value       = aws_lb.example.dns_name
  description = "The domain name of the load balancer"
}
```

You can then “pass through” this output in *stage/services/webserver-cluster/outputs.tf* and *prod/services/webserver-cluster/outputs.tf* as follows:

```
output "alb_dns_name" {
  value       = module.webserver_cluster.alb_dns_name
```

```
    description = "The domain name of the load balancer"
  }
```

Your web server cluster is almost ready to deploy. The only thing left is to take a few gotchas into account.

Module Gotchas

When creating modules, watch out for these gotchas:

- File paths
- Inline blocks

File Paths

In Chapter 3, you moved the User Data script for the web server cluster into an external file, *user-data.sh*, and used the `templatefile` built-in function to read this file from disk. The catch with the `templatefile` function is that the filepath you use must be a relative path (you don't want to use absolute file paths, as your Terraform code may run on many different computers, each with a different disk layout)—but what is it relative to?

By default, Terraform interprets the path relative to the current working directory. That works if you're using the `templatefile` 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 `templatefile` function in the root module), but that won't work when you're using `templatefile` in a module that's defined in a separate folder (a reusable module).

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` when calling the `templatefile` function in *modules/services/webserver-cluster/main.tf*:

```
user_data = templatefile("${path.module}/user-data.sh", {
  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. An *inline block* is an argument you set within a resource of the format:

```
resource "xxx" "yyy" {
  <NAME> {
    [CONFIG...]
  }
}
```

where `NAME` is the name of the inline block (e.g., `ingress`) and `CONFIG` consists of one or more arguments that are specific to that inline block (e.g., `from_port` and `to_port`). For example, with the `aws_security_group_resource`, you can define ingress and egress rules using either inline blocks (e.g., `ingress { … }`) or separate `aws_security_group_rule` resources.

If you try to use a mix of *both* inline blocks and separate resources, due to how Terraform is designed, you will get errors where the configurations conflict and overwrite one another. Therefore, you must use one or the other. Here's my advice: when creating a module, you should always prefer using separate resources.

The advantage of using separate resources is that they can be added anywhere, whereas an inline block can only be added within the module that creates a resource. So using solely separate resources makes your module more flexible and configurable.

For example, in the `webserver-cluster` module (*modules/services/webserver-cluster/main.tf*), you used inline blocks to define ingress and egress rules:

```
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
  }
}
```

With these inline blocks, a user of this module has no way to add additional ingress or egress rules from outside the module. To make your module more flexible, you should change it 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
}
```

You should also export the ID of the `aws_security_group` as an output variable in *modules/services/webserver-cluster/outputs.tf*:

```
output "alb_security_group_id" {
  value       = aws_security_group.alb.id
  description = "The ID of the Security Group attached to the load balancer
}
```

Now, if you needed to expose an extra port in just the staging environment (e.g., for testing), you can do this by adding an `aws_security_group_rule` resource to *stage/services/webserver-cluster/main.tf*:

```
module "webserver_cluster" {
  source = "../../../modules/services/webserver-cluster"

  # (parameters hidden for clarity)
}

resource "aws_security_group_rule" "allow_testing_inbound" {
  type              = "ingress"
  security_group_id = module.webserver_cluster.alb_security_group_id

  from_port   = 12345
  to_port     = 12345
  protocol    = "tcp"
  cidr_blocks = ["0.0.0.0/0"]
}
```

Had you defined even a single ingress or egress rule as an inline block, this code would not work. Note that this same type of problem affects a number of Terraform resources, such as the following:

- `aws_security_group` and `aws_security_group_rule`
- `aws_route_table` and `aws_route`
- `aws_network_acl` and `aws_network_acl_rule`

At this point, you are finally ready to deploy your web server cluster in both staging and production. Run `terraform apply` as usual, and enjoy using two separate copies of your infrastructure.

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.[1]

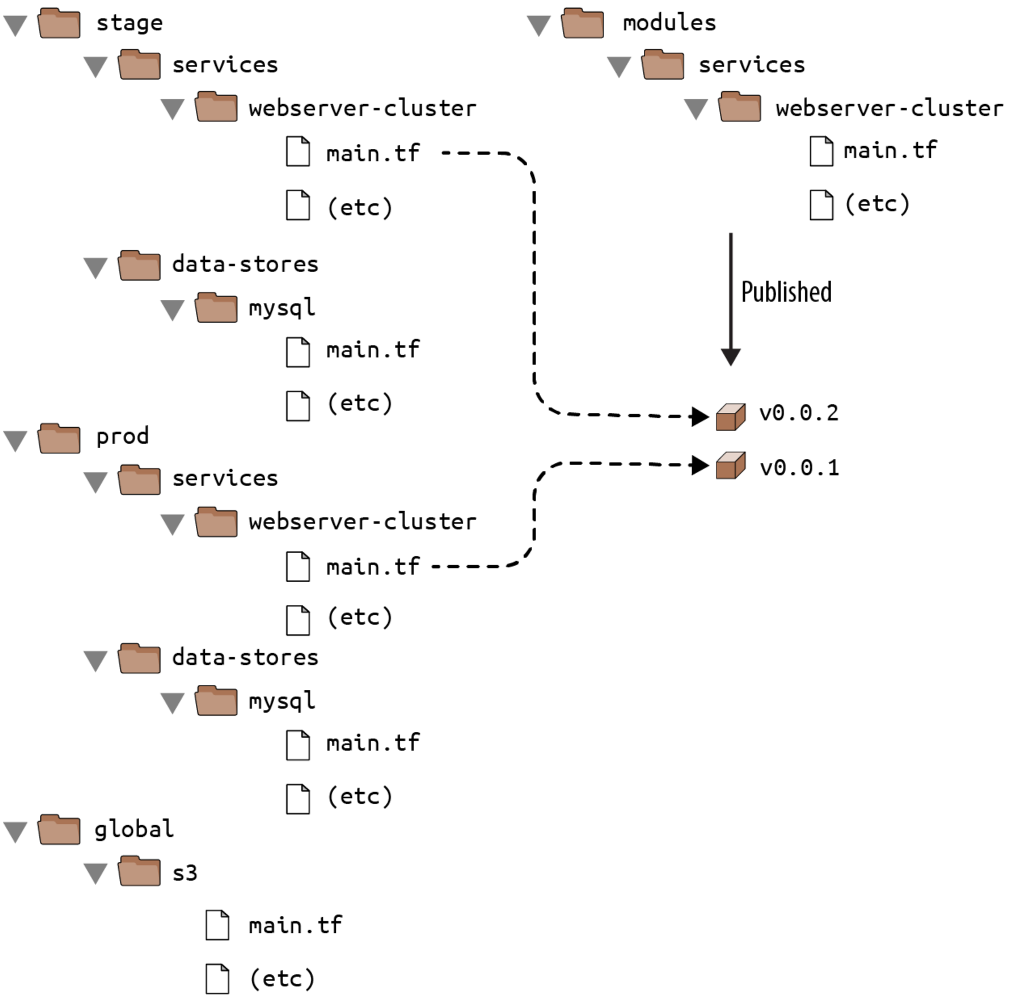

Figure 4-5. By versioning your modules, you can use different versions in different environments: e.g., v0.0.1 in prod and v0.0.2 in stage.

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.

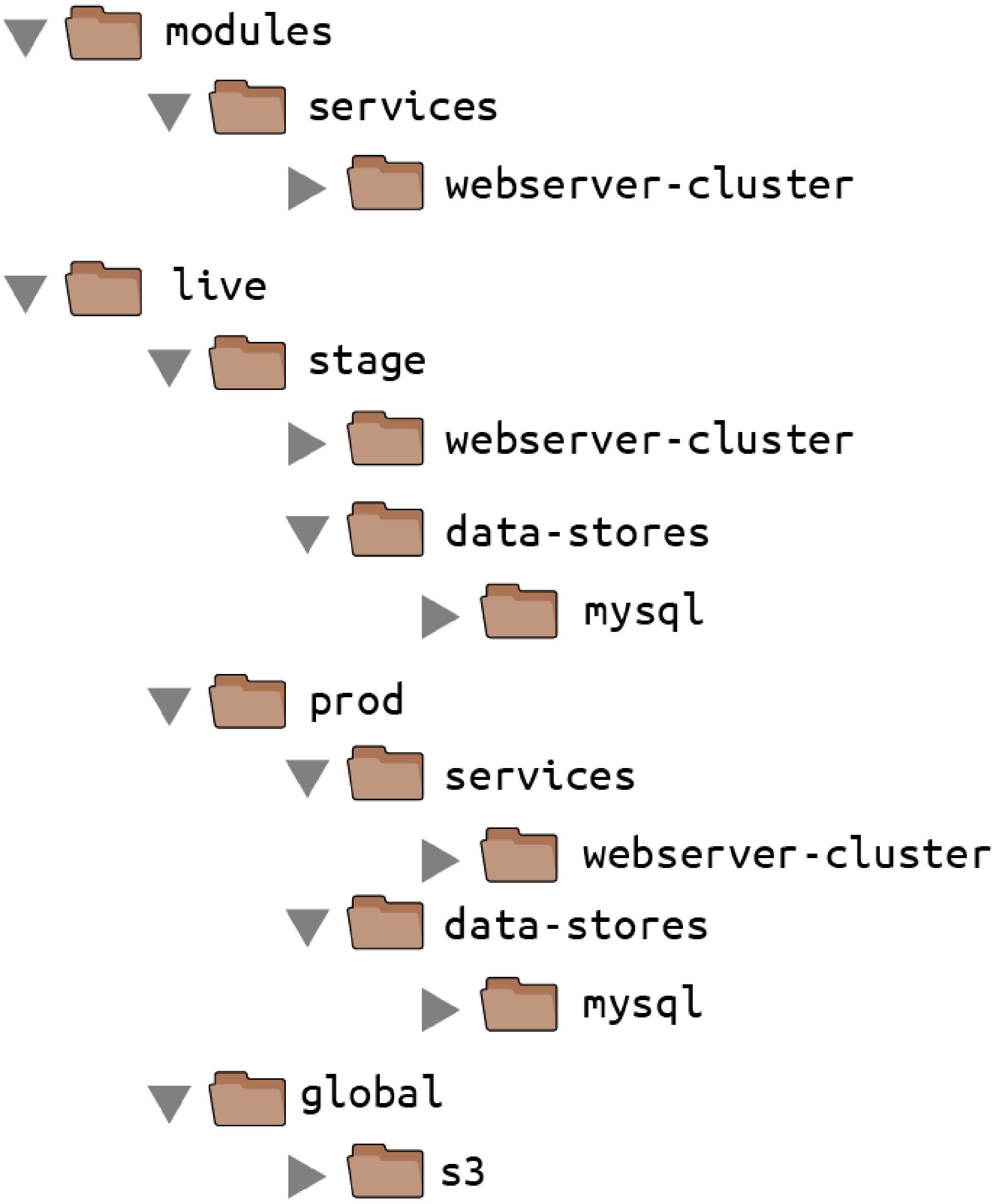

Figure 4-6. You should store reusable, versioned modules in one repo (modules) and the configuration for your live environments in another repo (live).

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           = "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 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 re-running `terraform init`:

```
$ terraform init
Initializing modules...
Downloading git@github.com:brikis98/terraform-up-and-running-code.git?ref=v
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.[2]

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           = "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
}
```

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           = "webservers-prod"
  db_remote_state_bucket = "(YOUR_BUCKET_NAME)"
  db_remote_state_key    = "prod/data-stores/mysql/terraform.tfstate"

  instance_type = "m4.large"
  min_size      = 2
  max_size      = 10
}
```

After v0.0.2 has been thoroughly tested and proven in staging, you can then update production, too. But if there turns out to be a bug in v0.0.2, no big deal, because it has no effect on the real users of your production environment. Fix the bug, release a new version, and repeat the entire process again until you have something stable enough for production.

DEVELOPING MODULES

Versioned modules are great when you're deploying to a shared environment (e.g., staging or production), but when you're just testing on your own computer, you'll want to use local file paths. This allows you to iterate faster, because you'll be able to make a change in the module folders and rerun the `plan` or `apply` command in the live folders immediately, rather than having to commit your code, publish a new version, and rerun `init` each time.

Since the goal of this book is to help you learn and experiment with Terraform as quickly as possible, the rest of the code examples will use local file paths for modules.

Conclusion

By defining infrastructure as code in modules, you can apply a variety of software engineering best practices to your infrastructure. You can validate each change to a module through code reviews and automated tests, you can create semantically versioned releases of each module, and you can safely try out different versions of a module in different environments and roll back to previous versions if you hit a problem.

All of this can dramatically increase your ability to build infrastructure quickly and reliably because developers will be able to reuse entire pieces of proven, tested, and documented infrastructure. For example, you could create a canonical module that defines how to deploy a single microservice—including how to run a cluster, how to scale the cluster in response to load, and how to distribute traffic requests across the cluster—and each team could use this module to manage their own microservices with just a few lines of code.

To make such a module work for multiple teams, the Terraform code in that module must be flexible and configurable. For example, one team might want to use your module to deploy a single Instance of their microservice with no load balancer, whereas another might want a dozen Instances of their microservice with a load balancer to distribute traffic between those Instances. How do you do conditional statements in Terraform? Is there a way to do a for-loop? Is there a way to use Terraform to roll out changes to this microservice without downtime? These advanced aspects of Terraform syntax are the topic of Chapter 5.

[1] For the full details on source URLs, see the Terraform website.

[2] See the GitHub documentation for a nice guide on working with SSH keys.