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-
```