- `aws_lb`
- `aws_lb_listener`
- `aws_security_group` (the one for the ALB but not for the Instances)
- `aws_security_group_rule` (both of the rules for the ALB but not the one for the Instances)

Create *modules/networking/alb/variables.tf*, and define a single variable within:

```
variable "alb_name" {
  description = "The name to use for this ALB"
  type        = string
}
```

Use this variable as the `name` argument of the `aws_lb` resource:

```
resource "aws_lb" "example" {
  name               = var.alb_name
  load_balancer_type = "application"
  subnets            = data.aws_subnets.default.ids
  security_groups    = [aws_security_group.alb.id]
}
```

And the `name` argument of the `aws_security_group` resource:

```
resource "aws_security_group" "alb" {
  name = var.alb_name
}
```

This is a lot of code to shuffle around, so feel free to use the code examples for this chapter from GitHub.

Composable Modules

You now have two small modules—`asg-rolling-deploy` and `alb`—that each do one thing and do it well. How do you make them work

together? How do you build modules that are reusable and composable? This question is not unique to Terraform but is something programmers have been thinking about for decades. To quote Doug McIlroy,[5] the original developer of Unix pipes and a number of other Unix tools, including `diff`, `sort`, `join`, and `tr`:

> *This is the Unix philosophy: Write programs that do one thing and do it well. Write programs to work together.*

One way to do this is through *function composition*, in which you can take the outputs of one function and pass them as the inputs to another. For example, if you had the following small functions in Ruby:

```
# Simple function to do addition
def add(x, y)
  return x + y
end

# Simple function to do subtraction
def sub(x, y)
  return x - y
end

# Simple function to do multiplication
def multiply(x, y)
  return x * y
end
```

you can use function composition to put them together by taking the outputs from `add` and `sub` and passing them as the inputs to `multiply`:

```
# Complex function that composes several simpler functions
def do_calculation(x, y)
  return multiply(add(x, y), sub(x, y))
end
```

One of the main ways to make functions composable is to minimize *side effects*: that is, where possible, avoid reading state from the outside world and instead have it passed in via input parameters, and avoid writing state to the outside world and instead return the result of your computations via

output parameters. Minimizing side effects is one of the core tenets of functional programming because it makes the code easier to reason about, easier to test, and easier to reuse. The reuse story is particularly compelling, since function composition allows you to gradually build up more complicated functions by combining simpler functions.

Although you can't avoid side effects when working with infrastructure code, you can still follow the same basic principles in your Terraform modules: pass everything in through input variables, return everything through output variables, and build more complicated modules by combining simpler modules.

Open up *modules/cluster/asg-rolling-deploy/variables.tf*, and add four new input variables:

```
variable "subnet_ids" {
  description = "The subnet IDs to deploy to"
  type        = list(string)
}

variable "target_group_arns" {
  description = "The ARNs of ELB target groups in which to
register Instances"
  type        = list(string)
  default     = []
}

variable "health_check_type" {
  description = "The type of health check to perform. Must be one
of: EC2, ELB."
  type        = string
  default     = "EC2"
}

variable "user_data" {
  description = "The User Data script to run in each Instance at
boot"
  type        = string
  default     = null
}
```

The first variable, `subnet_ids`, tells the `asg-rolling-deploy` module what subnets to deploy into. Whereas the `webserver-cluster` module was hardcoded to deploy into the Default VPC and subnets, by exposing the `subnet_ids` variable, you allow this module to be used with any VPC or subnets. The next two variables, `target_group_arns` and `health_check_type`, configure how the ASG integrates with load balancers. Whereas the `webserver-cluster` module had a built-in ALB, the `asg-rolling-deploy` module is meant to be a generic module, so exposing the load-balancer settings as input variables allows you to use the ASG with a wide variety of use cases; e.g., no load balancer, one ALB, multiple NLBs, and so on.

Take these three new input variables and pass them through to the `aws_autoscaling_group` resource in *modules/cluster/asg-rolling-deploy/main.tf*, replacing the previously hardcoded settings that were referencing resources (e.g., the ALB) and data sources (e.g., `aws_subnets`) that we didn't copy into the `asg-rolling-deploy` module:

```
resource "aws_autoscaling_group" "example" {
  name                 = var.cluster_name
  launch_configuration = aws_launch_configuration.example.name
  vpc_zone_identifier  = var.subnet_ids

  # Configure integrations with a load balancer
  target_group_arns    = var.target_group_arns
  health_check_type    = var.health_check_type

  min_size = var.min_size
  max_size = var.max_size

  # (...)
}
```

The fourth variable, `user_data`, is for passing in a User Data script. Whereas the `webserver-cluster` module had a hardcoded User Data script that could only be used to deploy a "Hello, World" app, by taking in a User Data script as an input variable, the `asg-rolling-deploy`

module can be used to deploy any app across an ASG. Pass this `user_data` variable through to the `aws_launch_configuration` resource:

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

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

You'll also want to add a couple of useful output variables to *modules/cluster/asg-rolling-deploy/outputs.tf*:

```
output "asg_name" {
  value       = aws_autoscaling_group.example.name
  description = "The name of the Auto Scaling Group"
}

output "instance_security_group_id" {
  value       = aws_security_group.instance.id
  description = "The ID of the EC2 Instance Security Group"
}
```

Outputting this data makes the `asg-rolling-deploy` module even more reusable, since consumers of the module can use these outputs to add new behaviors, such as attaching custom rules to the security group.

For similar reasons, you should add several output variables to *modules/networking/alb/outputs.tf*:

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

```
output "alb_http_listener_arn" {
  value       = aws_lb_listener.http.arn
  description = "The ARN of the HTTP listener"
}

output "alb_security_group_id" {
  value       = aws_security_group.alb.id
  description = "The ALB Security Group ID"
}
```

You'll see how to use these shortly.

The last step is to convert the `webserver-cluster` module into a `hello-world-app` module that can deploy a "Hello, World" app using the `asg-rolling-deploy` and `alb` modules. To do this, rename *module/services/webserver-cluster* to *module/services/hello-world-app*. After all the changes in the previous steps, you should have only the following resources and data sources left in *module/services/hello-world-app/main.tf*:

- `aws_lb_target_group`
- `aws_lb_listener_rule`
- `terraform_remote_state` (for the DB)
- `aws_vpc`
- `aws_subnets`

Add the following variable to *modules/services/hello-world-app/variables.tf*:

```
variable "environment" {
  description = "The name of the environment we're deploying to"
  type        = string
}
```

Now, add the `asg-rolling-deploy` module that you created earlier to the `hello-world-app` module to deploy an ASG:

```
module "asg" {
  source = "../../cluster/asg-rolling-deploy"

  cluster_name  = "hello-world-${var.environment}"
  ami           = var.ami
  instance_type = var.instance_type

  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
    server_text = var.server_text
  })

  min_size           = var.min_size
  max_size           = var.max_size
  enable_autoscaling = var.enable_autoscaling

  subnet_ids        = data.aws_subnets.default.ids
  target_group_arns = [aws_lb_target_group.asg.arn]
  health_check_type = "ELB"

  custom_tags = var.custom_tags
}
```

And add the `alb` module, also that you created earlier, to the `hello-world-app` module to deploy an ALB:

```
module "alb" {
  source = "../../networking/alb"

  alb_name   = "hello-world-${var.environment}"
  subnet_ids = data.aws_subnets.default.ids
}
```

Note the use of the input variable `environment` as a way to enforce a naming convention, so all of your resources will be namespaced based on the environment (e.g., `hello-world-stage`, `hello-world-prod`). This code also sets the new `subnet_ids`, `target_group_arns`, `health_check_type`, and `user_data` variables you added earlier to appropriate values.

Next, you need to configure the ALB target group and listener rule for this app. Update the `aws_lb_target_group` resource in *modules/services/hello-world-app/main.tf* to use `environment` in its `name`:

```
resource "aws_lb_target_group" "asg" {
  name     = "hello-world-${var.environment}"
  port     = var.server_port
  protocol = "HTTP"
  vpc_id   = data.aws_vpc.default.id

  health_check {
    path                = "/"
    protocol            = "HTTP"
    matcher             = "200"
    interval            = 15
    timeout             = 3
    healthy_threshold   = 2
    unhealthy_threshold = 2
  }
}
```

Now, update the `listener_arn` parameter of the `aws_lb_listener_rule` resource to point at the `alb_http_listener_arn` output of the ALB module:

```
resource "aws_lb_listener_rule" "asg" {
  listener_arn = module.alb.alb_http_listener_arn
  priority     = 100

  condition {
    path_pattern {
      values = ["*"]
    }
  }

  action {
    type             = "forward"
    target_group_arn = aws_lb_target_group.asg.arn
  }
}
```