two to boot up, and then use a web browser or a tool like `curl` to make an HTTP request to this IP address at port 8080:

```
$ curl http://<EC2_INSTANCE_PUBLIC_IP>:8080
Hello, World
```

Yay! You now have a working web server running in AWS!

NETWORK SECURITY

To keep all of the examples in this book simple, they deploy not only into your Default VPC (as mentioned earlier) but also into the default *subnets* of that VPC. A VPC is partitioned into one or more subnets, each with its own IP addresses. The subnets in the Default VPC are all *public subnets*, which means they get IP addresses that are accessible from the public internet. This is why you are able to test your EC2 Instance from your home computer.

Running a server in a public subnet is fine for a quick experiment, but in real-world usage, it's a security risk. Hackers all over the world are *constantly* scanning IP addresses at random for any weakness. If your servers are exposed publicly, all it takes is accidentally leaving a single port unprotected or running out-of-date code with a known vulnerability, and someone can break in.

Therefore, for production systems, you should deploy all of your servers, and certainly all of your data stores, in *private subnets*, which have IP addresses that can be accessed only from within the VPC and not from the public internet. The only servers you should run in public subnets are a small number of reverse proxies and load balancers that you lock down as much as possible (you'll see an example of how to deploy a load balancer later in this chapter).

Deploying a Configurable Web Server

You might have noticed that the web server code has the port 8080 duplicated in both the security group and the User Data configuration. This violates the *Don't Repeat Yourself (DRY)* principle: every piece of knowledge must have a single, unambiguous, authoritative representation within a system.[13] If you have the port number in two places, it's easy to update it in one place but forget to make the same change in the other place.

To allow you to make your code more DRY and more configurable, Terraform allows you to define *input variables*. Here's the syntax for declaring a variable:

```
variable "NAME" {
  [CONFIG ...]
}
```

The body of the variable declaration can contain the following optional parameters:

`description`

> It's always a good idea to use this parameter to document how a variable is used. Your teammates will be able to see this description not only while reading the code but also when running the `plan` or `apply` commands (you'll see an example of this shortly).

`default`

> There are a number of ways to provide a value for the variable, including passing it in at the command line (using the `-var` option), via a file (using the `-var-file` option), or via an environment variable (Terraform looks for environment variables of the name `TF_VAR_<variable_name>`). If no value is passed in, the variable will fall back to this default value. If there is no default value, Terraform will interactively prompt the user for one.

`type`

This allows you to enforce *type constraints* on the variables a user passes in. Terraform supports a number of type constraints, including `string`, `number`, `bool`, `list`, `map`, `set`, `object`, `tuple`, and `any`. It's always a good idea to define a type constraint to catch simple errors. If you don't specify a type, Terraform assumes the type is `any`.

`validation`

This allows you to define custom validation rules for the input variable that go beyond basic type checks, such as enforcing minimum or maximum values on a number. You'll see an example of validations in Chapter 8.

`sensitive`

If you set this parameter to `true` on an input variable, Terraform will not log it when you run `plan` or `apply`. You should use this on any secrets you pass into your Terraform code via variables: e.g., passwords, API keys, etc. I'll talk more about secrets in Chapter 6.

Here is an example of an input variable that checks to verify that the value you pass in is a number:

```
variable "number_example" {
  description = "An example of a number variable in Terraform"
  type        = number
  default     = 42
}
```

And here's an example of a variable that checks whether the value is a list:

```
variable "list_example" {
  description = "An example of a list in Terraform"
  type        = list
  default     = ["a", "b", "c"]
}
```

You can combine type constraints, too. For example, here's a list input variable that requires all of the items in the list to be numbers:

```
variable "list_numeric_example" {
  description = "An example of a numeric list in Terraform"
  type        = list(number)
  default     = [1, 2, 3]
}
```

And here's a map that requires all of the values to be strings:

```
variable "map_example" {
  description = "An example of a map in Terraform"
  type        = map(string)

  default = {
    key1 = "value1"
    key2 = "value2"
    key3 = "value3"
  }
}
```

You can also create more complicated *structural types* using the `object` type constraint:

```
variable "object_example" {
  description = "An example of a structural type in Terraform"
  type        = object({
    name    = string
    age     = number
    tags    = list(string)
    enabled = bool
  })

  default = {
    name    = "value1"
    age     = 42
    tags    = ["a", "b", "c"]
    enabled = true
  }
}
```

The preceding example creates an input variable that will require the value to be an object with the keys `name` (which must be a string), `age` (which must be a number), `tags` (which must be a list of strings), and `enabled` (which must be a Boolean). If you try to set this variable to a value that doesn't match this type, Terraform immediately gives you a type error. The following example demonstrates trying to set `enabled` to a string instead of a Boolean:

```
variable "object_example_with_error" {
  description = "An example of a structural type in Terraform
with an error"
  type        = object({
    name    = string
    age     = number
    tags    = list(string)
    enabled = bool
  })

  default = {
    name    = "value1"
    age     = 42
    tags    = ["a", "b", "c"]
    enabled = "invalid"
  }
}
```

You get the following error:

```
$ terraform apply

Error: Invalid default value for variable

  on variables.tf line 78, in variable
"object_example_with_error":
  78:   default = {
  79:     name    = "value1"
  80:     age     = 42
  81:     tags    = ["a", "b", "c"]
  82:     enabled = "invalid"
  83:   }

This default value is not compatible with the variable's type
```

```
constraint: a
bool is required.
```

Coming back to the web server example, what you need is a variable that stores the port number:

```
variable "server_port" {
  description = "The port the server will use for HTTP requests"
  type        = number
}
```

Note that the `server_port` input variable has no `default`, so if you run the `apply` command now, Terraform will interactively prompt you to enter a value for `server_port` and show you the `description` of the variable:

```
$ terraform apply

var.server_port
  The port the server will use for HTTP requests

  Enter a value:
```

If you don't want to deal with an interactive prompt, you can provide a value for the variable via the `-var` command-line option:

```
$ terraform plan -var "server_port=8080"
```

You could also set the variable via an environment variable named `TF_VAR_<name>`, where `<name>` is the name of the variable you're trying to set:

```
$ export TF_VAR_server_port=8080
$ terraform plan
```

And if you don't want to deal with remembering extra command-line arguments every time you run `plan` or `apply`, you can specify a `default` value:

```
variable "server_port" {
  description = "The port the server will use for HTTP requests"
  type        = number
  default     = 8080
}
```

To use the value from an input variable in your Terraform code, you can use a new type of expression called a *variable reference*, which has the following syntax:

```
var.<VARIABLE_NAME>
```

For example, here is how you can set the `from_port` and `to_port` parameters of the security group to the value of the `server_port` variable:

```
resource "aws_security_group" "instance" {
  name = "terraform-example-instance"

  ingress {
    from_port   = var.server_port
    to_port     = var.server_port
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

It's also a good idea to use the same variable when setting the port in the User Data script. To use a reference inside of a string literal, you need to use a new type of expression called an *interpolation*, which has the following syntax:

```
"${...}"
```

You can put any valid reference within the curly braces, and Terraform will convert it to a string. For example, here's how you can use `var.server_port` inside of the User Data string:

```
  user_data = <<-EOF
              #!/bin/bash
              echo "Hello, World" > index.xhtml
              nohup busybox httpd -f -p ${var.server_port} &
              EOF
```

In addition to input variables, Terraform also allows you to define *output variables* by using the following syntax:

```
output "<NAME>" {
  value = <VALUE>
  [CONFIG ...]
}
```

The `NAME` is the name of the output variable, and `VALUE` can be any Terraform expression that you would like to output. The `CONFIG` can contain the following optional parameters:

`description`

> It's always a good idea to use this parameter to document what type of data is contained in the output variable.

`sensitive`

> Set this parameter to `true` to instruct Terraform not to log this output at the end of `plan` or `apply`. This is useful if the output variable contains secrets such as passwords or private keys. Note that if your output variable references an input variable or resource attribute marked with `sensitive = true`, you are *required* to mark the output variable with `sensitive = true` as well to indicate you are intentionally outputting a secret.

`depends_on`

> Normally, Terraform automatically figures out your dependency graph based on the references within your code, but in rare situations, you have to give it extra hints. For example, perhaps you have an output variable that returns the IP address of a server, but that IP won't be

accessible until a security group (firewall) is properly configured for that server. In that case, you may explicitly tell Terraform there is a dependency between the IP address output variable and the security group resource using `depends_on`.

For example, instead of having to manually poke around the EC2 console to find the IP address of your server, you can provide the IP address as an output variable:

```
output "public_ip" {
  value       = aws_instance.example.public_ip
  description = "The public IP address of the web server"
}
```

This code uses an attribute reference again, this time referencing the `public_ip` attribute of the `aws_instance` resource. If you run the `apply` command again, Terraform will not apply any changes (because you haven't changed any resources), but it will show you the new output at the very end:

```
$ terraform apply

(...)

aws_security_group.instance: Refreshing state... [id=sg-
078ccb4f9533d2c1a]
aws_instance.example: Refreshing state... [id=i-
028cad2d4e6bddec6]

Apply complete! Resources: 0 added, 0 changed, 0 destroyed.

Outputs:

public_ip = "54.174.13.5"
```

As you can see, output variables show up in the console after you run `terraform apply`, which users of your Terraform code might find useful (e.g., you now know what IP to test after the web server is deployed).