Let's go through these one a time.

Provisioners

Terraform *provisioners* are used to execute scripts either on the local machine or a remote machine when you run Terraform, typically to do the work of bootstrapping, configuration management, or cleanup. There are several different kinds of provisioners, including `local-exec` (execute a script on the local machine), `remote-exec` (execute a script on a remote resource), and `file` (copy files to a remote resource).[7]

You can add provisioners to a resource by using a `provisioner` block. For example, here is how you can use the `local-exec` provisioner to execute a script on your local machine:

```
resource "aws_instance" "example" {
  ami           = data.aws_ami.ubuntu.id
  instance_type = "t2.micro"

  provisioner "local-exec" {
    command = "echo \"Hello, World from $(uname -smp)\""
  }
}
```

When you run `terraform apply` on this code, it prints "Hello, World from" and then the local operating system details using the `uname` command:

```
$ terraform apply

(...)

aws_instance.example (local-exec): Hello, World from Darwin
x86_64 i386

(...)

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

Trying out a `remote-exec` provisioner is a little more complicated. To execute code on a remote resource, such as an EC2 Instance, your Terraform client must be able to do the following:

Communicate with the EC2 Instance over the network

You already know how to allow this with a security group.

Authenticate to the EC2 Instance

The `remote-exec` provisioner supports SSH and WinRM connections.

Since the examples in this book have you launch Linux (Ubuntu) EC2 Instances, you'll want to use SSH authentication. And that means you'll need to configure SSH keys. Let's begin by creating a security group that allows inbound connections to port 22, the default port for SSH:

```
resource "aws_security_group" "instance" {
  ingress {
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"

    # To make this example easy to try out, we allow all SSH
connections.
    # In real world usage, you should lock this down to solely
trusted IPs.
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

With SSH keys, the normal process would be for you to generate an SSH key pair on your computer, upload the public key to AWS, and store the private key somewhere secure where your Terraform code can access it. However, to make it easier for you to try out this code, you can use a resource called `tls_private_key` to automatically generate a private key:

```
# To make this example easy to try out, we generate a private key
in Terraform.
# In real-world usage, you should manage SSH keys outside of
Terraform.
resource "tls_private_key" "example" {
  algorithm = "RSA"
  rsa_bits  = 4096
}
```

This private key is stored in Terraform state, which is not great for production use cases but is fine for this learning exercise. Next, upload the public key to AWS using the `aws_key_pair` resource:

```
resource "aws_key_pair" "generated_key" {
  public_key = tls_private_key.example.public_key_openssh
}
```

Finally, let's begin writing the code for the EC2 Instance:

```
data "aws_ami" "ubuntu" {
  most_recent = true
  owners      = ["099720109477"] # Canonical

  filter {
    name   = "name"
    values = ["ubuntu/images/hvm-ssd/ubuntu-focal-20.04-amd64-
server-*"]
  }
}

resource "aws_instance" "example" {
  ami                    = data.aws_ami.ubuntu.id
  instance_type          = "t2.micro"
  vpc_security_group_ids = [aws_security_group.instance.id]
  key_name               = aws_key_pair.generated_key.key_name
}
```

Just about all of this code should be familiar to you: it's using the `aws_ami` data source to find Ubuntu AMI and using the `aws_instance` resource to deploy that AMI on a `t2.micro` instance, associating that instance with the security group you created earlier. The only new item is the use of the `key_name` attribute in the `aws_instance` resource to

instruct AWS to associate your public key with this EC2 Instance. AWS will add that public key to the server's *authorized_keys* file, which will allow you to SSH to that server with the corresponding private key.

Next, add the `remote-exec` provisioner to the `aws_instance` resource:

```
resource "aws_instance" "example" {
  ami                    = data.aws_ami.ubuntu.id
  instance_type          = "t2.micro"
  vpc_security_group_ids = [aws_security_group.instance.id]
  key_name               = aws_key_pair.generated_key.key_name

  provisioner "remote-exec" {
    inline = ["echo \"Hello, World from $(uname -smp)\""]
  }
}
```

This looks nearly identical to the `local-exec` provisioner, except you use an `inline` argument to pass a list of commands to execute, instead of a single `command` argument. Finally, you need to configure Terraform to use SSH to connect to this EC2 Instance when running the `remote-exec` provisioner. You do this by using a `connection` block:

```
resource "aws_instance" "example" {
  ami                    = data.aws_ami.ubuntu.id
  instance_type          = "t2.micro"
  vpc_security_group_ids = [aws_security_group.instance.id]
  key_name               = aws_key_pair.generated_key.key_name

  provisioner "remote-exec" {
    inline = ["echo \"Hello, World from $(uname -smp)\""]
  }

  connection {
    type        = "ssh"
    host        = self.public_ip
    user        = "ubuntu"
    private_key = tls_private_key.example.private_key_pem
  }
}
```

This `connection` block tells Terraform to connect to the EC2 Instance's public IP address using SSH with `"ubuntu"` as the username (this is the default username for the root user on Ubuntu AMIs) and the autogenerated private key. If you run `terraform apply` on this code, you'll see the following:

```
$ terraform apply

(...)

aws_instance.example: Creating...
aws_instance.example: Still creating... [10s elapsed]
aws_instance.example: Still creating... [20s elapsed]
aws_instance.example: Provisioning with 'remote-exec'...
aws_instance.example (remote-exec): Connecting to remote host via
SSH...
aws_instance.example (remote-exec): Connecting to remote host via
SSH...
aws_instance.example (remote-exec): Connecting to remote host via
SSH...

(... repeats a few more times ...)

aws_instance.example (remote-exec): Connecting to remote host via
SSH...
aws_instance.example (remote-exec): Connected!
aws_instance.example (remote-exec): Hello, World from Linux
x86_64 x86_64

Apply complete! Resources: 4 added, 0 changed, 0 destroyed.
```

The `remote-exec` provisioner doesn't know exactly when the EC2 Instance will be booted and ready to accept connections, so it will retry the SSH connection multiple times until it succeeds or hits a timeout (the default timeout is five minutes, but you can configure it). Eventually, the connection succeeds, and you get a "Hello, World" from the server.

Note that, by default, when you specify a provisioner, it is a *creation-time provisioner*, which means that it runs (a) during `terraform apply`, and (b) only during the initial creation of a resource. The provisioner will *not* run on any subsequent calls to `terraform apply`, so creation-time

provisioners are mainly useful for running initial bootstrap code. If you set the `when = destroy` argument on a provisioner, it will be a *destroy-time provisioner*, which will run after you run `terraform destroy`, just before the resource is deleted.

You can specify multiple provisioners on the same resource and Terraform will run them one at a time, in order, from top to bottom. You can use the `on_failure` argument to instruct Terraform how to handle errors from the provisioner: if set to `"continue"`, Terraform will ignore the error and continue with resource creation or destruction; if set to `"abort"`, Terraform will abort the creation or destruction.