And whenever you want to see changes your teammates have made, you can *pull* them from `origin`:

```
git pull origin main
```

As you go through the rest of this book, and as you use Terraform in general, make sure to regularly `git commit` and `git push` your changes. This way, you'll not only be able to collaborate with team members on this code, but all of your infrastructure changes will also be captured in the commit log, which is very handy for debugging. You'll learn more about using Terraform as a team in Chapter 10.

Deploying a Single Web Server

The next step is to run a web server on this Instance. The goal is to deploy the simplest web architecture possible: a single web server that can respond to HTTP requests, as shown in Figure 2-6.

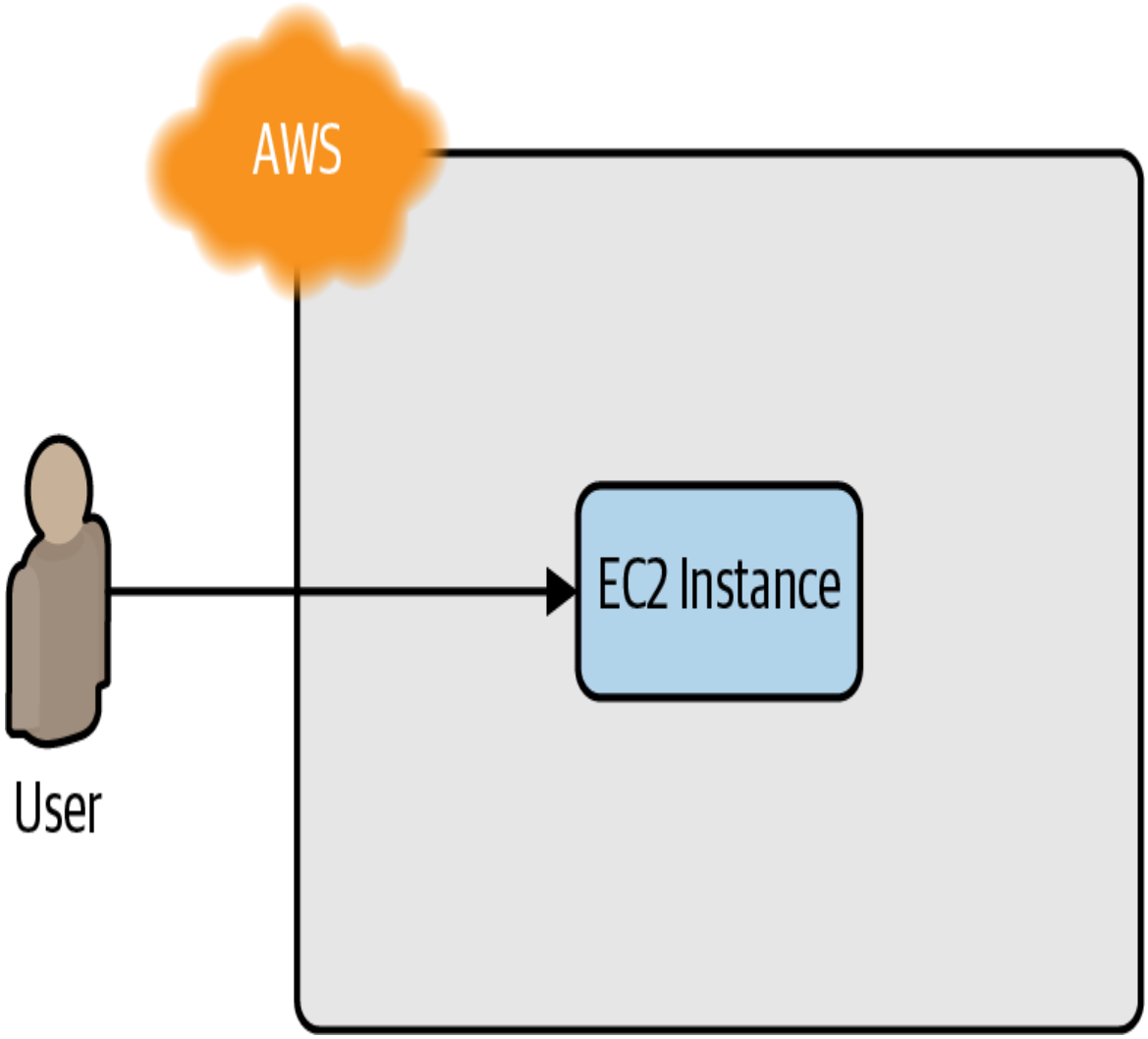

Figure 2-6. Start with a simple architecture: a single web server running in AWS that responds to HTTP requests.

In a real-world use case, you'd probably build the web server using a web framework like Ruby on Rails or Django, but to keep this example simple, let's run a dirt-simple web server that always returns the text "Hello, World":[10]

```
#!/bin/bash
echo "Hello, World" > index.xhtml
nohup busybox httpd -f -p 8080 &
```

This is a Bash script that writes the text "Hello, World" into *index.xhtml* and runs a tool called `busybox` (which is installed by default on Ubuntu) to

fire up a web server on port 8080 to serve that file. I wrapped the `busybox` command with `nohup` and an ampersand (`&`) so that the web server runs permanently in the background, whereas the Bash script itself can exit.

> **PORT NUMBERS**
>
> The reason this example uses port 8080, rather than the default HTTP port 80, is that listening on any port less than 1024 requires root user privileges. This is a security risk since any attacker who manages to compromise your server would get root privileges, too.
>
> Therefore, it's a best practice to run your web server with a non-root user that has limited permissions. That means you have to listen on higher-numbered ports, but as you'll see later in this chapter, you can configure a load balancer to listen on port 80 and route traffic to the high-numbered ports on your server(s).

How do you get the EC2 Instance to run this script? Normally, as discussed in "Server Templating Tools", you would use a tool like Packer to create a custom AMI that has the web server installed on it. Since the dummy web server in this example is just a one-liner that uses `busybox`, you can use a plain Ubuntu 20.04 AMI and run the "Hello, World" script as part of the EC2 Instance's *User Data* configuration. When you launch an EC2 Instance, you have the option of passing either a shell script or cloud-init directive to User Data, and the EC2 Instance will execute it during its very first boot. You pass a shell script to User Data by setting the `user_data` argument in your Terraform code as follows:

```
resource "aws_instance" "example" {
  ami                    = "ami-0fb653ca2d3203ac1"
  instance_type          = "t2.micro"

  user_data = <<-EOF
              #!/bin/bash
              echo "Hello, World" > index.xhtml
              nohup busybox httpd -f -p 8080 &
              EOF

  user_data_replace_on_change = true
```

```
  tags = {
    Name = "terraform-example"
  }
}
```

Two things to notice about the preceding code:

- The `<<-EOF` and `EOF` are Terraform's *heredoc* syntax, which allows you to create multiline strings without having to insert `\n` characters all over the place.
- The `user_data_replace_on_change` parameter is set to `true` so that when you change the `user_data` parameter and run `apply`, Terraform will terminate the original instance and launch a totally new one. Terraform's default behavior is to update the original instance in place, but since User Data runs only on the very first boot, and your original instance already went through that boot process, you need to force the creation of a new instance to ensure your new User Data script actually gets executed.

You need to do one more thing before this web server works. By default, AWS does not allow any incoming or outgoing traffic from an EC2 Instance. To allow the EC2 Instance to receive traffic on port 8080, you need to create a *security group*:

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

  ingress {
    from_port   = 8080
    to_port     = 8080
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

This code creates a new resource called `aws_security_group` (notice how all resources for the AWS Provider begin with `aws_`) and specifies that this group allows incoming TCP requests on port 8080 from the CIDR

block 0.0.0.0/0. *CIDR blocks* are a concise way to specify IP address ranges. For example, a CIDR block of 10.0.0.0/24 represents all IP addresses between 10.0.0.0 and 10.0.0.255. The CIDR block 0.0.0.0/0 is an IP address range that includes all possible IP addresses, so this security group allows incoming requests on port 8080 from any IP.[11]

Simply creating a security group isn't enough; you need to tell the EC2 Instance to actually use it by passing the ID of the security group into the `vpc_security_group_ids` argument of the `aws_instance` resource. To do that, you first need to learn about Terraform *expressions*.

An expression in Terraform is anything that returns a value. You've already seen the simplest type of expressions, *literals*, such as strings (e.g., `"ami-0fb653ca2d3203ac1"`) and numbers (e.g., `5`). Terraform supports many other types of expressions that you'll see throughout the book.

One particularly useful type of expression is a *reference*, which allows you to access values from other parts of your code. To access the ID of the security group resource, you are going to need to use a *resource attribute reference*, which uses the following syntax:

```
<PROVIDER>_<TYPE>.<NAME>.<ATTRIBUTE>
```

where `PROVIDER` is the name of the provider (e.g., `aws`), `TYPE` is the type of resource (e.g., `security_group`), `NAME` is the name of that resource (e.g., the security group is named `"instance"`), and `ATTRIBUTE` is either one of the arguments of that resource (e.g., `name`) or one of the attributes *exported* by the resource (you can find the list of available attributes in the documentation for each resource). The security group exports an attribute called `id`, so the expression to reference it will look like this:

```
aws_security_group.instance.id
```

You can use this security group ID in the `vpc_security_group_ids` argument of the `aws_instance` as follows:

```
resource "aws_instance" "example" {
  ami                    = "ami-0fb653ca2d3203ac1"
  instance_type          = "t2.micro"
  vpc_security_group_ids = [aws_security_group.instance.id]

  user_data = <<-EOF
              #!/bin/bash
              echo "Hello, World" > index.xhtml
              nohup busybox httpd -f -p 8080 &
              EOF

  user_data_replace_on_change = true

  tags = {
    Name = "terraform-example"
  }
}
```

When you add a reference from one resource to another, you create an *implicit dependency*. Terraform parses these dependencies, builds a dependency graph from them, and uses that to automatically determine in which order it should create resources. For example, if you were deploying this code from scratch, Terraform would know that it needs to create the security group before the EC2 Instance, because the EC2 Instance references the ID of the security group. You can even get Terraform to show you the dependency graph by running the `graph` command:

```
$ terraform graph

digraph {
        compound = "true"
        newrank = "true"
        subgraph "root" {
                "[root] aws_instance.example"
                  [label = "aws_instance.example", shape = "box"]
                "[root] aws_security_group.instance"
                  [label = "aws_security_group.instance", shape =
"box"]
                "[root] provider.aws"
                  [label = "provider.aws", shape = "diamond"]
                "[root] aws_instance.example" ->
                  "[root] aws_security_group.instance"
                "[root] aws_security_group.instance" ->
                  "[root] provider.aws"
```

```
                "[root] meta.count-boundary (EachMode fixup)" ->
                  "[root] aws_instance.example"
                "[root] provider.aws (close)" ->
                  "[root] aws_instance.example"
                "[root] root" ->
                  "[root] meta.count-boundary (EachMode fixup)"
                "[root] root" ->
                  "[root] provider.aws (close)"
        }
}
```

The output is in a graph description language called DOT, which you can turn into an image, similar to the dependency graph shown in Figure 2-7, by using a desktop app such as Graphviz or web app like GraphvizOnline.[12]

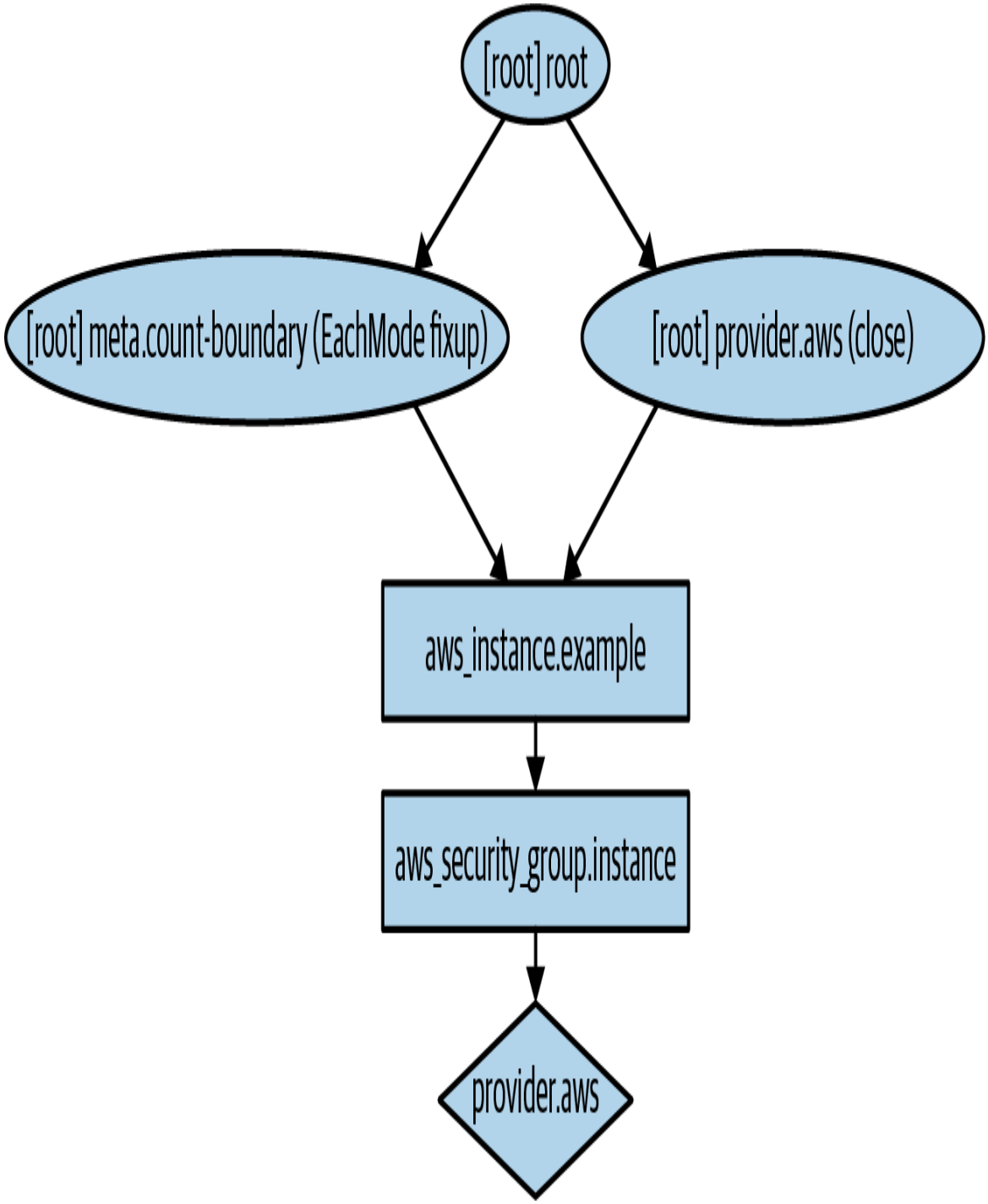

Figure 2-7. This is what the dependency graph for the EC2 Instance and its security group looks like when rendered with Graphviz.

When Terraform walks your dependency tree, it creates as many resources in parallel as it can, which means that it can apply your changes fairly efficiently. That's the beauty of a declarative language: you just specify

what you want, and Terraform determines the most efficient way to make it happen.

If you run the `apply` command, you'll see that Terraform wants to create a security group and replace the EC2 Instance with a new one that has the new user data:

```
$ terraform apply

(...)

Terraform will perform the following actions:

  # aws_instance.example must be replaced
-/+ resource "aws_instance" "example" {
        ami                           = "ami-0fb653ca2d3203ac1"
      ~ availability_zone             = "us-east-2c" -> (known
after apply)
      ~ instance_state                = "running" -> (known after
apply)
        instance_type                 = "t2.micro"
        (...)
      + user_data                     = "c765373..." # forces
replacement
      ~ volume_tags                   = {} -> (known after apply)
      ~ vpc_security_group_ids        = [
          - "sg-871fa9ec",
        ] -> (known after apply)
        (...)
    }

  # aws_security_group.instance will be created
  + resource "aws_security_group" "instance" {
      + arn                    = (known after apply)
      + description            = "Managed by Terraform"
      + egress                 = (known after apply)
      + id                     = (known after apply)
      + ingress                = [
          + {
              + cidr_blocks      = [
                  + "0.0.0.0/0",
                ]
              + description      = ""
              + from_port        = 8080
              + ipv6_cidr_blocks = []
              + prefix_list_ids  = []
```

```
            + protocol         = "tcp"
            + security_groups  = []
            + self             = false
            + to_port          = 8080
          },
      ]
    + name                   = "terraform-example-instance"
    + owner_id               = (known after apply)
    + revoke_rules_on_delete = false
    + vpc_id                 = (known after apply)
  }

Plan: 2 to add, 0 to change, 1 to destroy.

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value:
```

The `-/+` in the `plan` output means "replace"; look for the text "forces replacement" in the plan output to figure out what is forcing Terraform to do a replacement. Since you set `user_data_replace_on_change` to `true` and changed the `user_data` parameter, this will force a replacement, which means that the original EC2 Instance will be terminated and a completely new Instance will be created. This is an example of the immutable infrastructure paradigm discussed in "Server Templating Tools". It's worth mentioning that while the web server is being replaced, any users of that web server would experience downtime; you'll see how to do a zero-downtime deployment with Terraform in Chapter 5.

Since the plan looks good, enter **`yes`**, and you'll see your new EC2 Instance deploying, as shown in Figure 2-8.

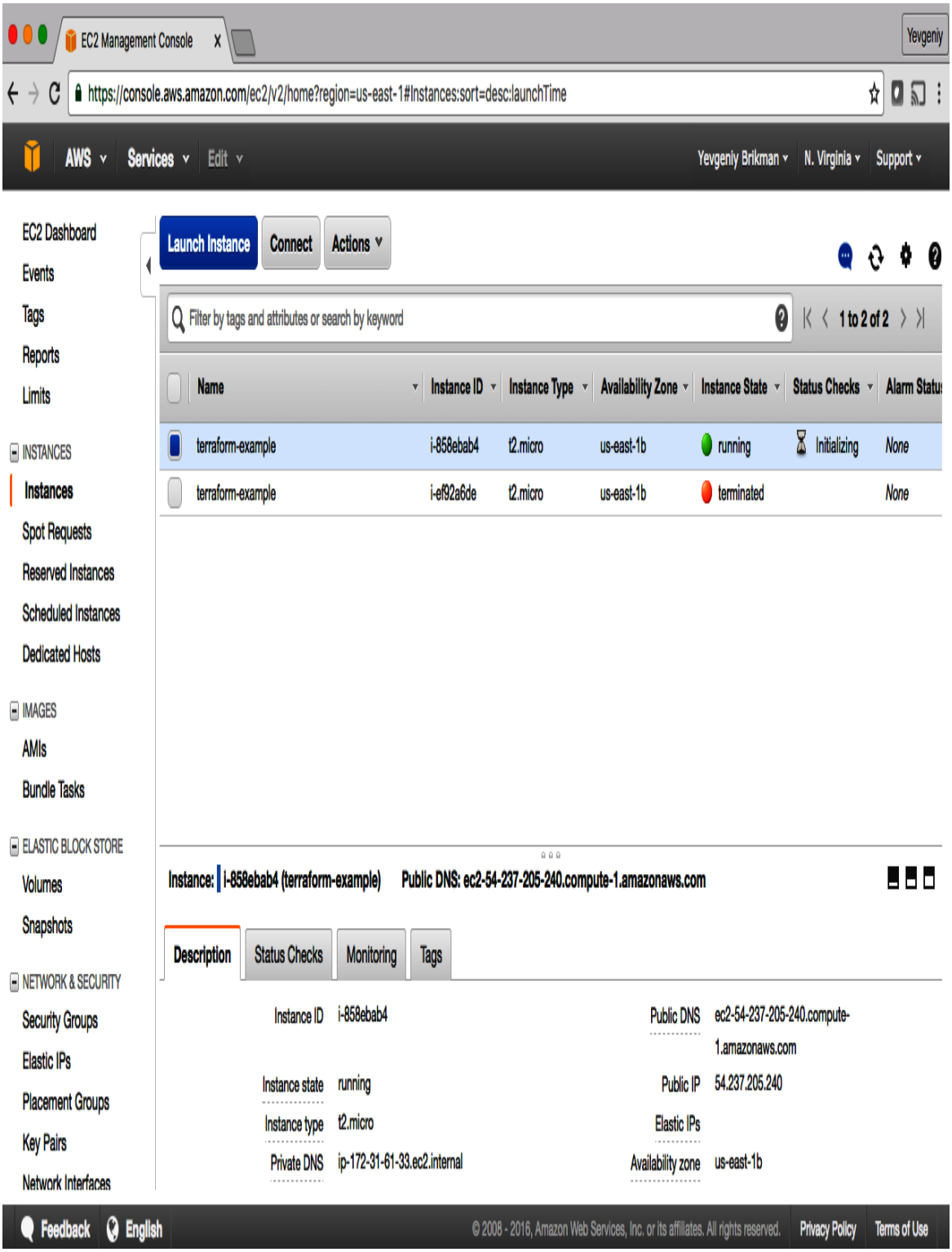

Figure 2-8. The new EC2 Instance with the web server code replaces the old Instance.

If you click your new Instance, you can find its public IP address in the description panel at the bottom of the screen. Give the Instance a minute or