Terraform

In the previous exercise we have discussed how to set up an infrastructure manually. For the purposes of this lecture Terraform is our tool of choice when it comes to cloud automation. There are, of course, many other open source tools like Ansible, or provider-dependent tools like AWS CloudFormation. We specifically chose Terraform because it is provider-independent and can be used to provision cloud resources with a wide range of providers.

You can download Terraform to your own computer. As it is a program written in Go you can simply unpack it and use it. We recommend adding it to your `PATH` environment variable for easy access. Since Terraform is capable of using multiple files we also recommend using an IDE or an editor with a directory listing function. The Intellij IDEA Community Edition has a plugin for Terraform that also offers code completion.

At it's core Terraform reads all `*.tf` files in a directory and merges them together. This provides an efficient way of structuring a project.

Each Terraform file contains a number of `data` and `resource` sections which describe the cloud environment. For example, you could create the Exoscale instance from the previous exercise using the following Terraform code:

```
data "exoscale_compute_template" "ubuntu" {
  zone = "at-vie-1"
  name = "Linux Ubuntu 20.04 LTS 64-bit"
}

resource "exoscale_compute" "mymachine" {
  zone         = "at-vie-1"
  display_name = "test"
  template_id  = data.exoscale_compute_template.ubuntu.id
  size         = "Micro"
  disk_size    = 10
  key_pair     = ""

  affinity_groups = []
  security_groups = ["default"]

  user_data = <<EOF
#!/bin/bash
set -e
apt update
apt install -y nginx
EOF
}
```

This exercise will guide you through the basics of using Terraform to provision cloud servers.

Tip

The entire source code for this exercise is available on GitHub.

Setting up the Exoscale provider

Before you can begin you will need to configure your cloud provider. In our case that will be Exoscale, so our first piece of code in a file called `provider.tf` will be as follows:

```
terraform {
  required_providers {
    exoscale = {
      source  = "terraform-providers/exoscale"
    }
  }
}

provider "exoscale" {
  key = "EXO..."
  secret = "..."
}
```

This configures the Exoscale provider. We can make sure the provider loads correctly by running `terraform init` in the directory of the project.

> **Tip**
>
> You need to run `terraform init` every time the provider configuration changes.

The above example is not ideal as it contains hard-coded credentials. As any code Terraform should be stored in a code versioning system such as Git so having hard-coded credentials is not ideal. Instead, let's create a separate variable section:

```
variable "exoscale_key" {
  description = "The Exoscale API key"
  type = string
}
variable "exoscale_secret" {
  description = "The Exoscale API secret"
  type = string
}
provider "exoscale" {
  key = var.exoscale_key
  secret = "${var.exoscale_secret}"
}
```

As you can see variables and other resources can be referenced in Terraform. This is important as it lets us write reusable and modular cloud manifests. Variables can be referenced in two formats: either by directly writing their name, or by including them in a string with the dollar sign.

Creating a virtual machine

Unlike in our previous exercise, we will take a little more elaborate route and actually create our own security group this time. We start by creating a file called `sg.tf` to hold the following security group configuration:

```
resource "exoscale_security_group" "sg" {
  name = "exercise-2"
}
```

We will add the rules in a separate step:

```
resource "exoscale_security_group_rule" "http" {
  security_group_id = exoscale_security_group.sg.id
  type = "INGRESS"
  protocol = "tcp"
  cidr = "0.0.0.0/0"
  start_port = 80
  end_port = 80
}
```

Once you have this small amount of code complete you can try and run the Terraform config:

```
terraform apply
```

This step will query you for the variables:

```
$ terraform apply
var.exoscale_key
  The Exoscale API key

  Enter a value: EXO...

var.exoscale_secret
  The Exoscale API secret

  Enter a value: ...
```

If you wish to automate this step you can create a file called `terraform.tfvars` with the following content:

```
exoscale_key = "EXO..."
exoscale_secret = "..."
```

After inserting the variables Terraform will present you with a plan. It is very important that you always read this plan as it is a list of things Terraform intends to do. This may very well include destroying and re-creating a server! In our case the plan looks like this:

```
Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # exoscale_security_group.sg will be created
  + resource "exoscale_security_group" "sg" {
      + id   = (known after apply)
      + name = "exercise-2"
    }

  # exoscale_security_group_rule.http will be created
  + resource "exoscale_security_group_rule" "http" {
      + cidr                = "0.0.0.0/0"
      + end_port            = 80
      + id                  = (known after apply)
```

```
      + protocol            = "tcp"
      + security_group      = (known after apply)
      + security_group_id   = (known after apply)
      + start_port          = 80
      + type                = "INGRESS"
      + user_security_group = (known after apply)
    }

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

This looks good, so you can enter "yes" to have Terraform execute the plan.

```
exoscale_security_group.sg: Creating...
exoscale_security_group.sg: Creation complete after 1s [id=beac3948-dc6f-44a9-89cb-
d3ff8ea1c319]
exoscale_security_group_rule.http: Creating...
exoscale_security_group_rule.http: Creation complete after 3s [id=33b86aea-
a64e-4708-82c8-43e88242f73f]

Apply complete! Resources: 2 added, 0 changed, 0 destroyed.
```

Sweet! Now, on to creating a virtual machine:

```
data "exoscale_compute_template" "ubuntu" {
  zone = "at-vie-1"
  name = "Linux Ubuntu 20.04 LTS 64-bit"
}

resource "exoscale_compute" "mymachine" {
  zone         = "at-vie-1"
  display_name = "test"
  template_id  = data.exoscale_compute_template.ubuntu.id
  size         = "Micro"
  disk_size    = 10
  key_pair     = ""

  affinity_groups = []
  security_groups = [exoscale_security_group.sg.name]

  user_data = <<EOF
#!/bin/bash
set -e
apt update
apt install -y nginx
EOF
}
```

As you can see we are referencing the previously-created security groups by name. This reference is important as Terraform attempts to execute resource creation *in parallel* and uses variables to determine which resources depend on each other. In other words, if you do not use a variable reference but hard-code the name Terraform will not be able to execute the instructions in the correct order.

Tip

You can force Terraform to destroy and recreate a resource by using `terraform taint exoscale_compute.ubuntu`.

About your state file

You may have noticed that a new file called `terraform.tfstate` appeared in your project folder. This file is *very* important as it contains links to the cloud resources you created. If you lose it Terraform will not know what servers it already created and attempt to re-create everything. Also, **do not put this file in Git!** It contains sensitive information like passwords!

Destroying resources

Once you are done with this exercise please run `terraform destroy` to tear down the infrastructure you just created.