Infrastructure automation and configuration with Terraform & Ansible

Dimitra Zuccarelli, 20072495

April 2019

Contents

1 Introduction

With the rise of Cloud Computing and on-demand self service provision of hardware resources, infrastructure has become increasingly dynamic. Resources can be scaled up and down to handle peaks in traffic throughout the day, without the upfront costs of depreciating hardware. Compared to traditional infrastructure management (where a single virtual machine may have lived for months to years), resources might only live for weeks or even days. This elasticity inherent to Cloud Computing has led to the need for automation of manual, repetitive tasks, in a reliable, non-human error prone way [2].

Infrastructure as Code is a solution to this problem, which allows DevOps teams to rapidly and reliably spin up their environments through automation of infrastructure, through code. Infrastructure is managed in the same way that other applications or code is managed by teams (Version control, Continuous Deployment, Continuous Delivery, etc) [1].

1.1 Terraform

Terraform is an open source tool developed by HashiCorp, which allows you to declaratively spin up infrastructure. It has support for Cloud providers such as AWS, GCP, Microsoft Azure and more. Once the Infrastructure as Code is written (using a high-level configuration language called HCL), Terraform generates an execution plan, which describes what needs to be done to reach the desired state. The plan is then executed to build the infrastructure. The Terraform files can be versioned and re-used to build repeatable environments.

Listing 1: Example of creating a VPC in HCL

```
// Default VPC definition
resource "aws_vpc" "default" {
  cidr_block         = "20.0.0.0/16"
  enable_dns_hostnames = true

  tags {
    Name = "haproxy_test_vpc"
  }
}
```

1.2 Ansible

Ansible is another IaC tool by Red Hat which can be used for environment configuration and application deployment. Ansible maps tasks grouped in *roles* to a set of hosts, which are described in an *inventory* file. Roles are declaratively described in YAML files.

Ansible is a popular choice for configuration management because, unlike other similar tools like Puppet or Chef, no additional software needs to be installed or running on the nodes for Ansible to execute playbooks, since it does this over SSH.

Listing 2: Example inventory file

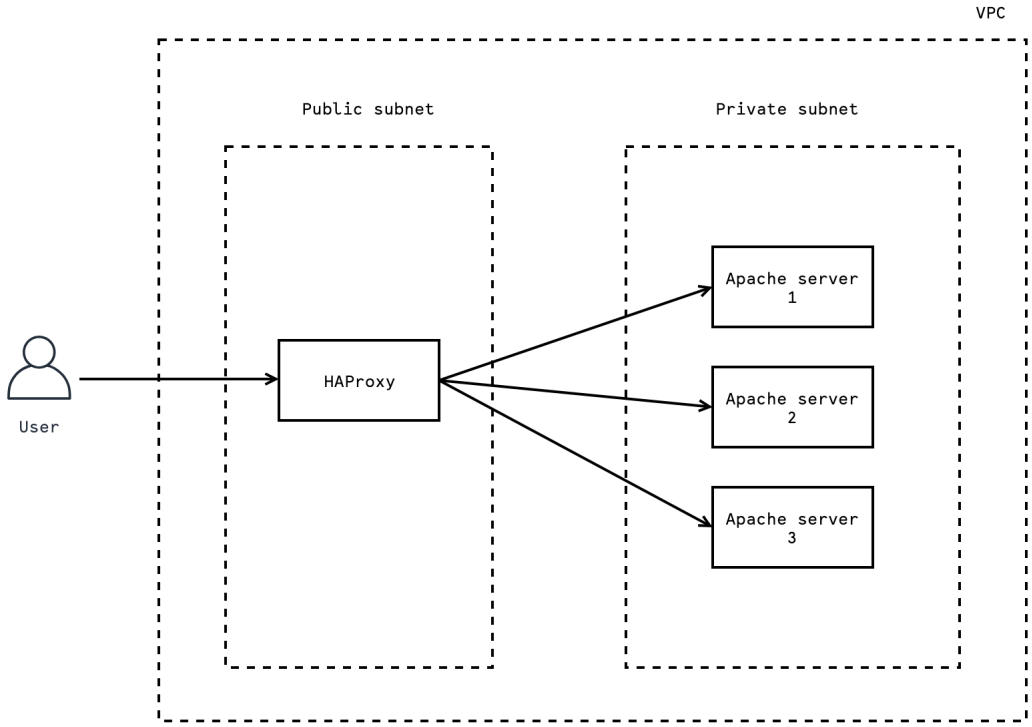
```
[loadbalancers]
10.0.0.1
10.0.0.2

[webservers]
10.0.0.3
10.0.0.4
10.0.0.5
```

Listing 3: Example task in YAML

```
- name: Copy the html file
  template:
    src: index.html.j2
    dest: /var/www/html/index.html
    mode: 0664
    owner: apache
    group: apache
```

2 Practical

An HAProxy load balancer will created to round robin across a cluster of backend web servers. For security, the web servers will be deployed in a private subnet in AWS, allowing only the HAProxy instance to access them. The HAProxy service will be deployed into a public subnet and accessible to the end user. This will all be automated using Ansible and Terraform. The code can be found at the link in Appendix A.



The AWS resources will be created using Terraform. The Terraform script creates the following:

- A Virtual Private Cloud (VPC) with a public subnet and a private subnet
- An EC2 instance for the HAProxy load balancer. This will use a custom HAProxy AMI based on the Amazon Linux AMI, although HAProxy could easily be installed with Ansible alternatively
- Three Amazon Linux EC2 instances for the web servers that will be load balanced across

Once the environment is spun up, it will be configured using Ansible. The Ansible playbook contains roles to:

- Install an Apache web server on each node in the private subnet. A custom HTML page is copied over which displays the IP address of the node, in order to visualise the load balancing
- Update the HAProxy configuration to load balance across the Apache web servers. To avoid manually adding the IP addresses of these nodes to the Ansible inventory, this will be done dynamically.

2.1 Terraform

The Terraform script is based on the script provided in the lab, with the difference of creating a private subnet which the web server nodes are deployed into. The HAProxy instance is deployed using a custom AMI, which simply installs and runs HAProxy on an Amazon Linux image using the `yum` package manager.

Listing 4: HAProxy AMI

```
data "aws_ami" "haproxy_aws_amis" {
  most_recent = true

  filter {
    name   = "name"
    values = ["HAProxy image"]
  }
  owners = ["155290810877"]
}
```

Listing 5: Private subnet

```
resource "aws_subnet" "tf_private_subnet" {
  vpc_id     = "${aws_vpc.default.id}"
  cidr_block = "20.0.2.0/24"

  tags {
    Name = "haproxy_private_subnet"
  }
}
```

Once the private subnet is created, a NAT gateway, route table and route table association also need to be created:

Listing 6: Create the NAT gateway

```
// Define our Nat Gateway
resource "aws_nat_gateway" "ng" {
  allocation_id = "${var.allocation_id}"
  subnet_id     = "${aws_subnet.tf_private_subnet.id}"

  tags {
    Name = "haproxy_test_ng"
  }
}

// Define private routing table
resource "aws_route_table" "r_private" {
  vpc_id = "${aws_vpc.default.id}"

  route {
    cidr_block = "0.0.0.0/0"
    gateway_id = "${aws_nat_gateway.ng.id}"
  }

  tags {
    Name = "haproxy_test_route_table_private"
  }
}

// Routing table association for private subnet
```

```
resource "aws_route_table_association" "a_private" {
  subnet_id     = "${aws_subnet.tf_private_subnet.id}"
  route_table_id = "${aws_route_table.r_private.id}"
}
```

Finally, the web servers can be deployed to the private subnet:

Listing 7: Deploying the web nodes into the private subnet

```
// Instance definition for Web backends
// Variable instance count
resource "aws_instance" "web_node" {
  count = "${var.web_cluster_size}"

  instance_type = "${var.aws_web_instance_type}"
  ami           = "${var.ami}"
  key_name      = "${var.key_name}"

  vpc_security_group_ids = ["${aws_security_group.web_node_sg.id}"]
  subnet_id              = "${aws_subnet.tf_private_subnet.id}"

  ...

  tags {
    Name = "haproxy_web_node"
  }
}
```

2.2 Ansible

Once the AWS environment was up and running, the biggest difficulty was attempting to SSH into the web server nodes (located in the private subnet) through a bastion host. Because I was struggling to do this with Ansible, I decided to try first with regular SSH config files. This part probably took the longest of the entire practical, since I couldn't get it working and found it very hard to debug. After a *very long time*, I realised I was using the wrong user (`ec2_user` instead of `ec2-user`). This is the final working SSH configuration file:

Listing 8: SSH Config file

```
Host bastion
  User ec2-user
  Hostname 34.254.144.101
  IdentityFile /Users/mitch/Desktop/ec2_ubuntu.pem

Host 20.0.2.*
  User ec2-user
  ProxyCommand ssh -W %h:%p bastion
  IdentityFile /Users/mitch/Desktop/ec2_ubuntu.pem
```

This can be saved in `~/.ssh/config` and detected automatically by Ansible, or Ansible can be configured directly to use it:

Listing 9: `group_vars/bastion.yml` file with ssh configuration

```
bastion_ip: "34.254.144.101"
user: "ec2-user"

ansible_ssh_common_args: '-o ProxyCommand="ssh -i /Users/mitch/Desktop/ec2_ubuntu.pem -W %h:%p
    -q {{ user }}@{{ bastion_ip }}"'
```

In order to automatically detect the IP addresses of the web server nodes, Ansible provide a `ec2.py` script and an `ec2.ini` inventory file which can automatically detect nodes in your AWS account. I couldn't get the `ec2.py` script working and eventually realised this was because I was using Python3 and the script was only compatible with Python2. I found a pull request (https://github.com/ansible/ansible/issues/18312) which adds support for Python3, which fixed the problem. Next, the `ec2.ini` file needed to be updated to include instances which do not have public IP addresses:

```
vpc_destination_variable = private_ip_address
```

Once this was done, the `httpd` role could be applied to every node which is tagged with `"haproxy_web_node"`, by adding the `"tag_Name_"` prefix:

Listing 10: site.yml

```
- name: Configure the backend servers
  hosts: tag_Name_haproxy_web_node
  become: true
  roles:
    - httpd
```

Unfortunately, because changing the `vpc_destination_variable` outputs only private IP addresses, the public IP address for the HAProxy instance needed to be added to a static inventory file:

Listing 11: Static inventory file

```
[haproxy]
34.254.144.181
```

Listing 12: site.yml

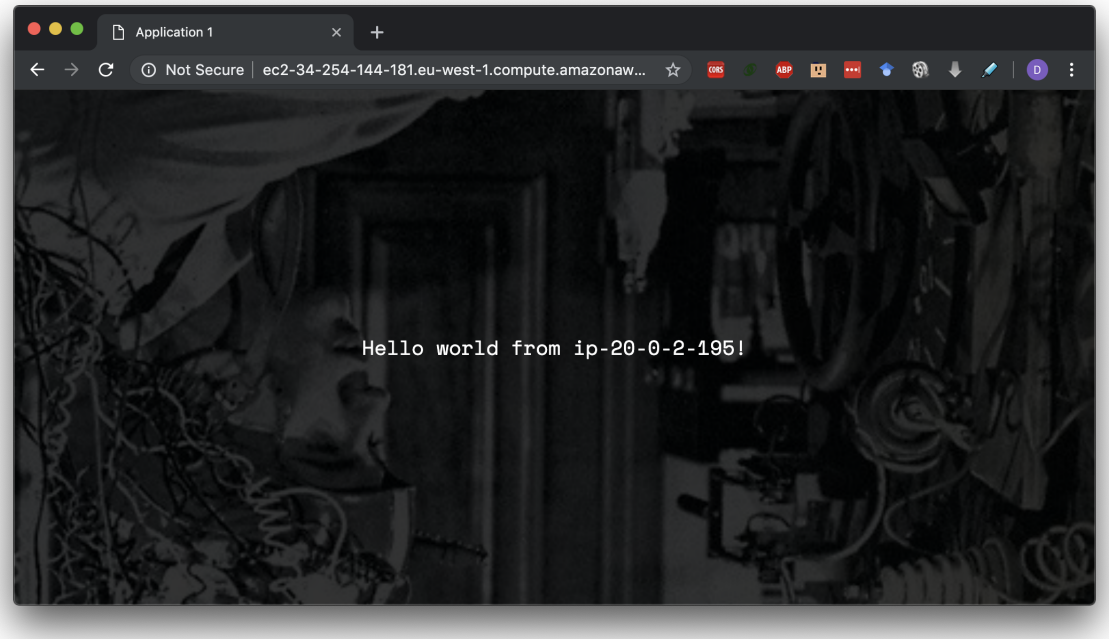
```
- name: Configure the haproxy load balancer
  hosts: haproxy
  become: true
  roles:
    - haproxy
```

Finally, the dynamic inventory can be used in the HAProxy configuration template, under the `backend` section:

Listing 13: HAProxy configuration template

```
backend app
    balance     roundrobin
{% for backend in groups['tag_Name_haproxy_web_node'] %}
    server      {{ hostvars[backend]['ec2_private_dns_name'] }} {{
        hostvars[backend]['ec2_private_ip_address'] }}:80 cookie {{
        hostvars[backend]['ec2_private_dns_name'] }} check
{% endfor %}
```

Heading to the load balancer's public IP address/public DNS name shows the working configuration:



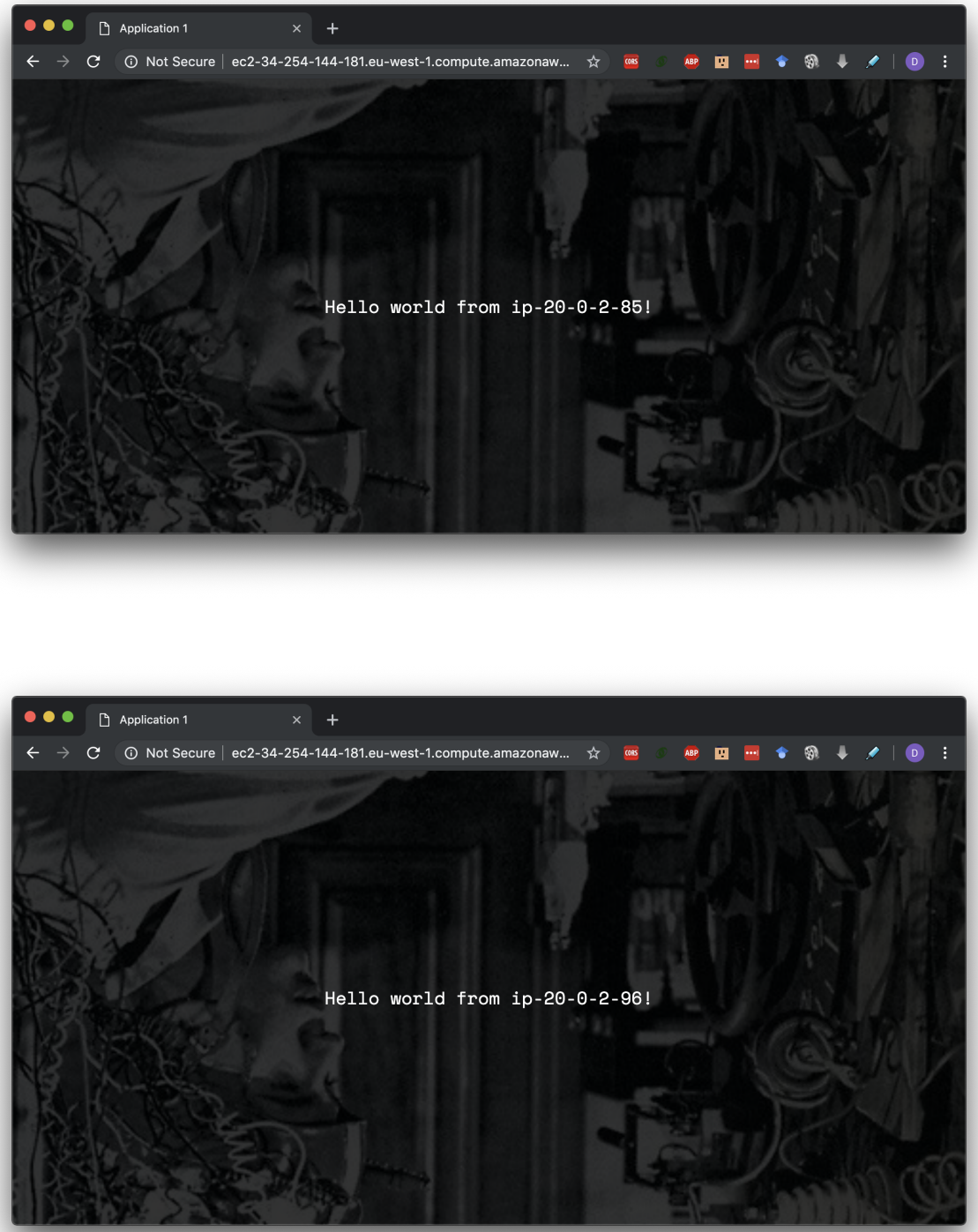


Figure 1: Load Balancing

3 Conclusions

I found the lab to be interesting and definitely see the importance of using Infrastructure as Code within an organisation. Having a set of Ansible or Terraform scripts means that DevOps teams can consistently and repeatedly spin up the same environment with much less time and effort. Additionally, changes can be tracked in version control and thoroughly documented, which makes it easy to rollback or keep track of changes to the infrastructure in general.

I found the HCL syntax to be easy to follow despite never having seen it before, although the Terraform scripts can get unwieldy. I imagine this is the most difficult aspect for large organisations with large, complex infrastructures to manage, as onboarding could be quite tough for new employees. In the long run, however, I don't think it would be sustainable for companies *not* to adopt an IaC approach.

I used HAProxy and Ansible for my first ever term paper for Cloud Computing in Second year, so I found this very nice to circle back to, with a few more years experience and knowledge. I definitely found it easier to get to grips with the syntax and the file structure of Ansible in general and of the HAProxy configuration files, although I have used them quite a bit since. Overall, for my own Final Year Project, I have found Ansible to be an indispensable tool which has saved me significant amounts of time. On a large scale, it is easy to see how much more so tools like Ansible and Terraform are needed.

Appendices

A Github Repository

https://github.com/dimitraz/aws-terraform-ansible

References

[1] Sam Guckenheimer. What is infrastructure as code? - azure devops.

[2] HashiCorp. What is infrastructure as code and why is it important?, Dec 2018.