Project Cloud Computing WS 2020/21

Infrastructure as Code (with Terraform)

OR

HashiCorp
Terraform

12.02.2021

Submitted by: Gökhan Yildirim, Samir Hamiani, Victoria Chaikovska

Frankfurt University of Applied Sciences

Faculty of Computer Science and Engineering

[goekhany@stud.fra-uas.de, samir.hamiani@stud.fra-uas.de, chaikovs@stud.fra-uas.de] [16]

Content

1. Infrastructure as a code……………………………………………………..2
2. Konzept, Introduction of the Infrastructure……………………..….6
3. Creating AWS-Account / Set-up……………………………………..7
4. Terraform installation and configuration…………………………..11
5. Development of Terraform Script…………………………………..17
6. Ansible installation and configuration……………………………...25
7. Development of Ansible Script……………………………………...29
8. Comparing Ansible and Terraform………………………………....37
9. Summary……………………………………………………………………..42

1. Infrastructure as Code

In the current times, more and more companies rely on Infrastructure as a Service (IaaS) solutions to implement and deploy automatically working environments, entire IT systems or even Cloud infrastructures. Infrastructure as Code provides IT infrastructure services such as computing power, storage and networking programmed into machine-readable code, in a similar way to software [8] .

Infrastructure as Code is an approach to describe and manage resources. There are 2 types of IaC tools :

- configuration management tools (e.g Ansible, Chef, Puppet)
- orchestration (e.g Terraform, CloudFormation)

Configuration management tools are aimed at modifying the existing infrastructure, which is close to procedural approach [9]. Some of the configuration tools are also overlapped with orchestration. Orchestration tools work declaratively. They describe the infrastructure and keep its state.

If a change is not applicable by an orchestration tool (e.g some settings of EC2 instances or DynamoDB indexes with existing data), it may require complete recreation of the resource and it's possible, because it keeps the entire configuration of the infrastructure.

Some of the advantages of Iac are:

- simplicity: deploying entire infrastructure with script.
- efficiency and speed: easily for different environments.
- low risk: reduced risk of errors.
- Costs: automated processes reduce time.

Terraform

Terraform is IaC engine which allows infrastructures to be developed, modified and versioned securely and efficiently on various types of providers: On-premise, AWS, Azure, Google, Kubernetes, etc [10]. At the same time some of the concrete services from some providers might be not supported or verified by Terraform. The figure 1 represents a functionality overview of Terraform.

One of the reasons Terraform is becoming more popular is because it has a simple syntax that allows for simple modularity and works well with multi-cloud systems. Another feature for using Terraform is managing infrastructure-as-a-code, which is also a foundation for DevOps practices such as version control, code review, continuous integration and continuous deployment.

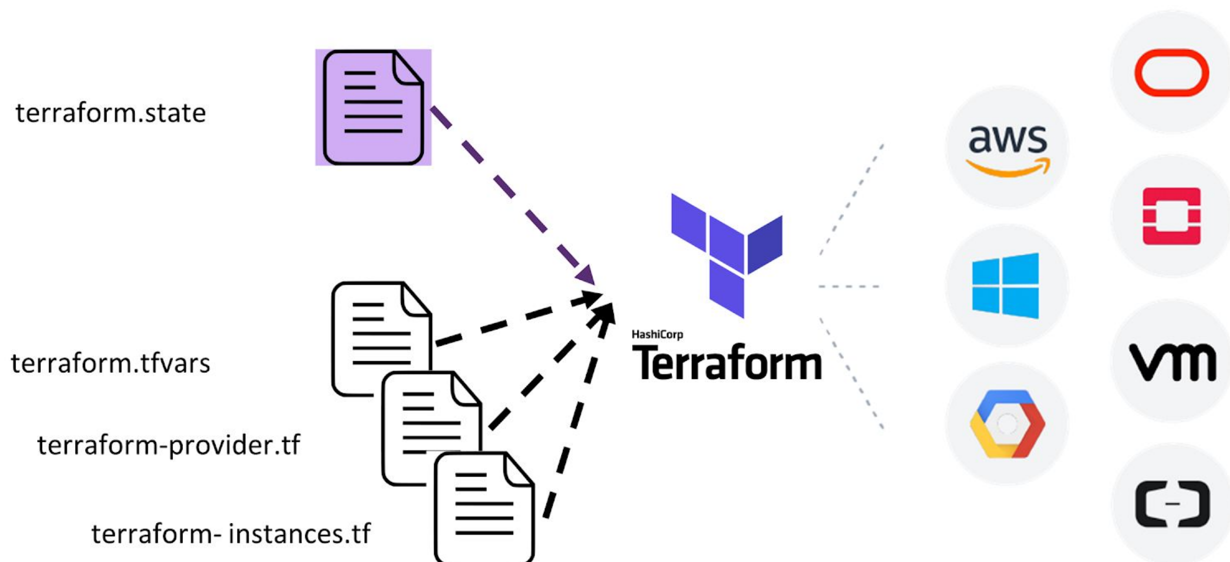
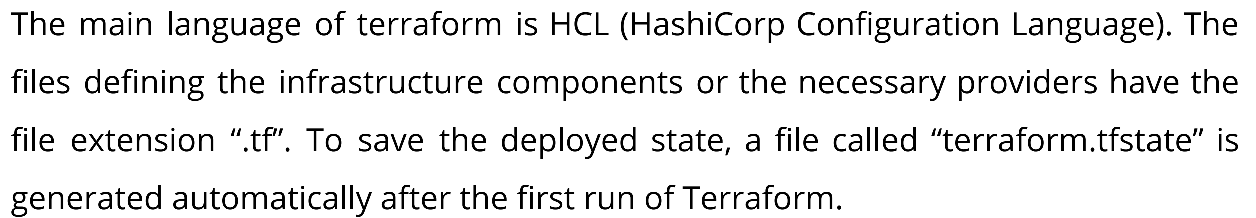


Figure 1: Terraform [12]



The main language of terraform is HCL (HashiCorp Configuration Language). The files defining the infrastructure components or the necessary providers have the file extension ".tf". To save the deployed state, a file called "terraform.tfstate" is generated automatically after the first run of Terraform.

While execution terraform scans the current directory for configuration file. If there is no configuration file Terraform produces a configuration file [11]. Terraform can detect changes and create, modify, and destroy infrastructure resources to match the desired state in the configuration file.

It is possible to manage many popular providers like AWS, Azure, Google and Kubernetes with Terraform.

Ansible

Ansible is an Open-Source tool for providing infrastructure as code. Ansible allows automatic provisioning configuration management and Infrastructure orchestration [14]. Ansible configure slave nodes, which are connected via ssh to the master. The slave nodes are managed in the inventory list (also named hostfile). The figure 2 below shows the Ansible architecture.

The Configurations of the slaves are done with the Use of Ansible modules. The ansible modules are written in the language YAML. Ansible Modules include a routine of tasks, which have a special use. The modules can be executed in the console or in Playbooks. Ansible Playbooks serve as a manual and describe the commands (Plays) to achieve the desired state. This state can be basic settings, but also a complete setup.

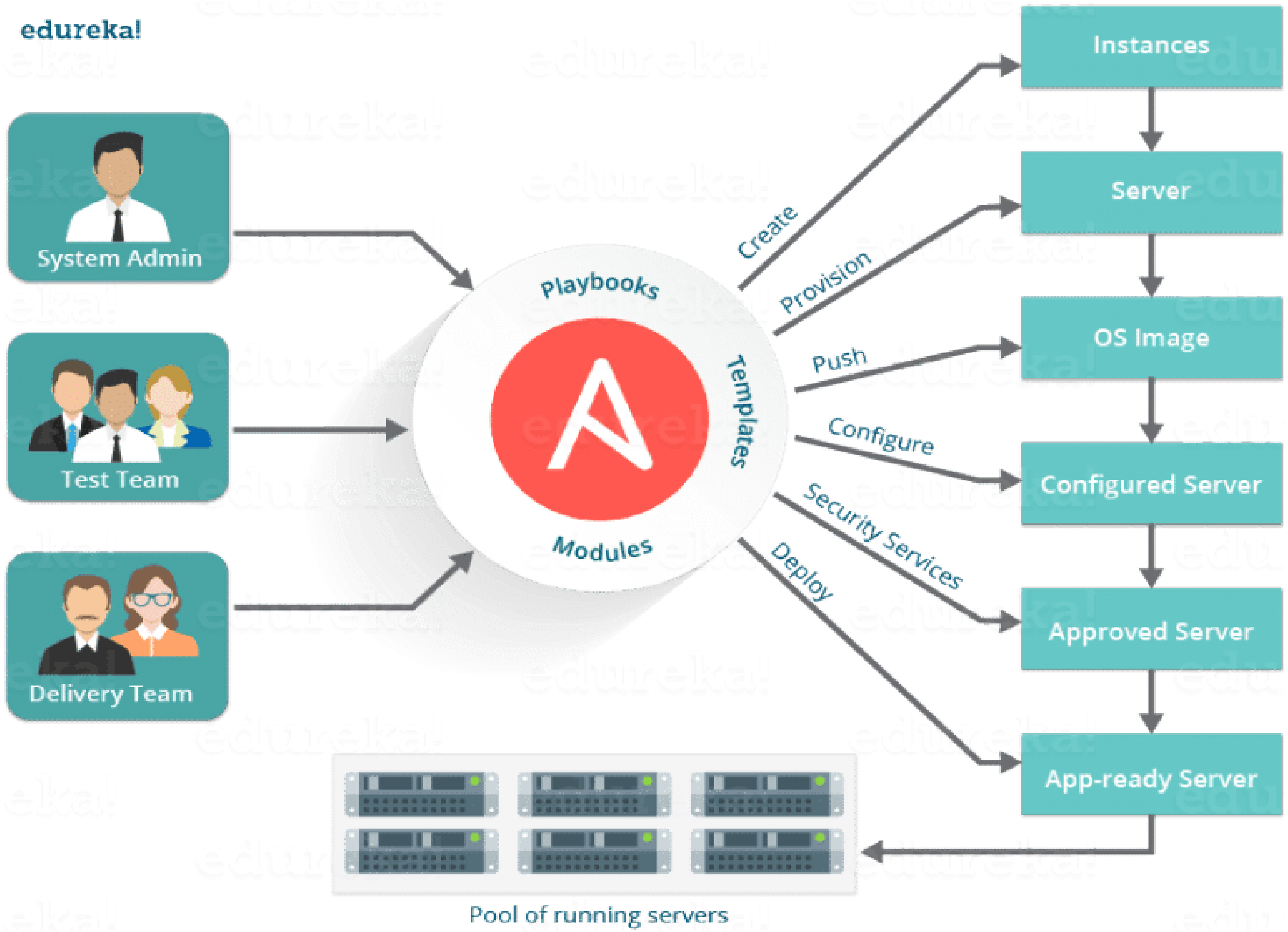


Figure 2: Ansible [13]

2.Konzept, Introduction of the Infrastructure

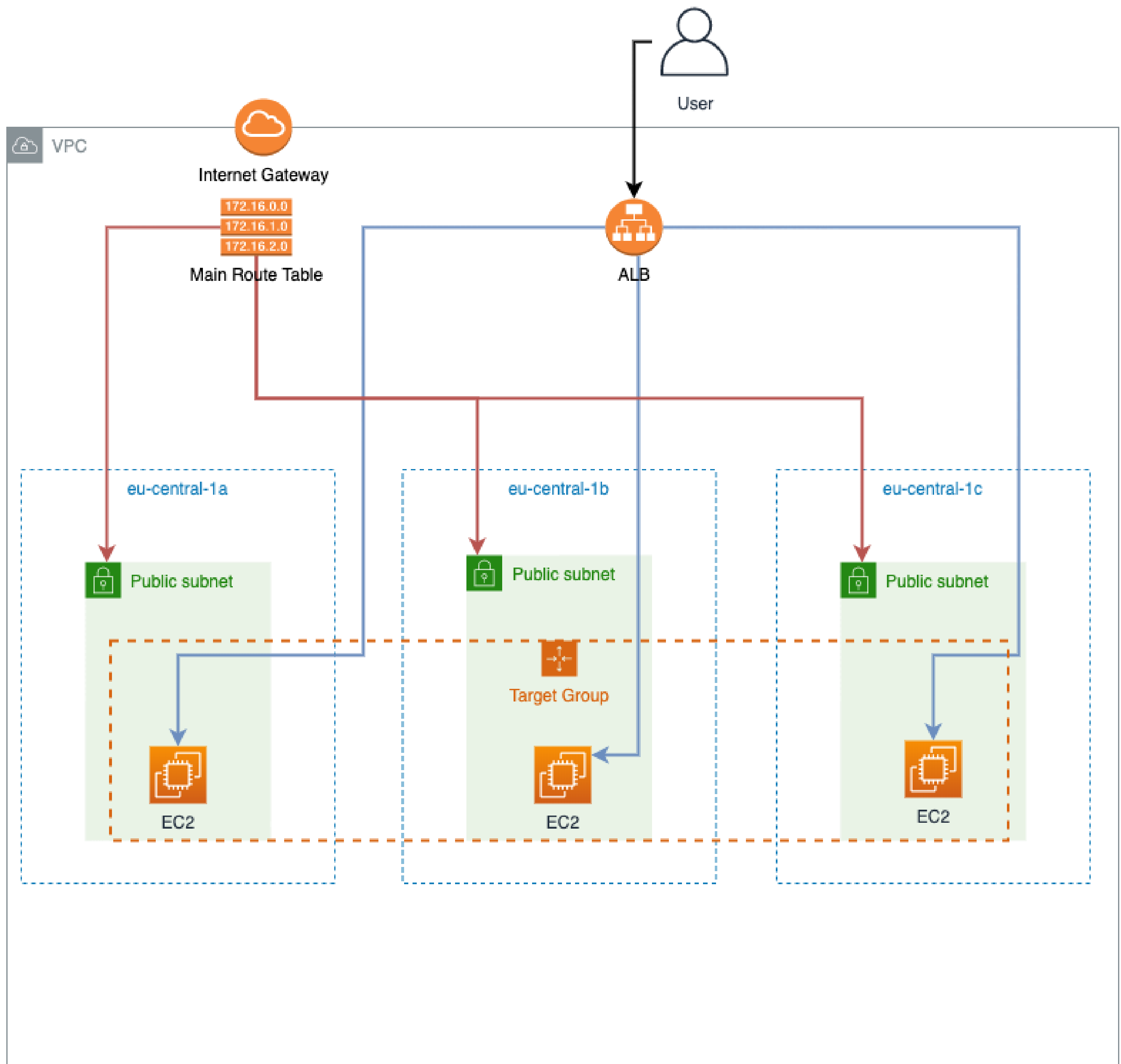
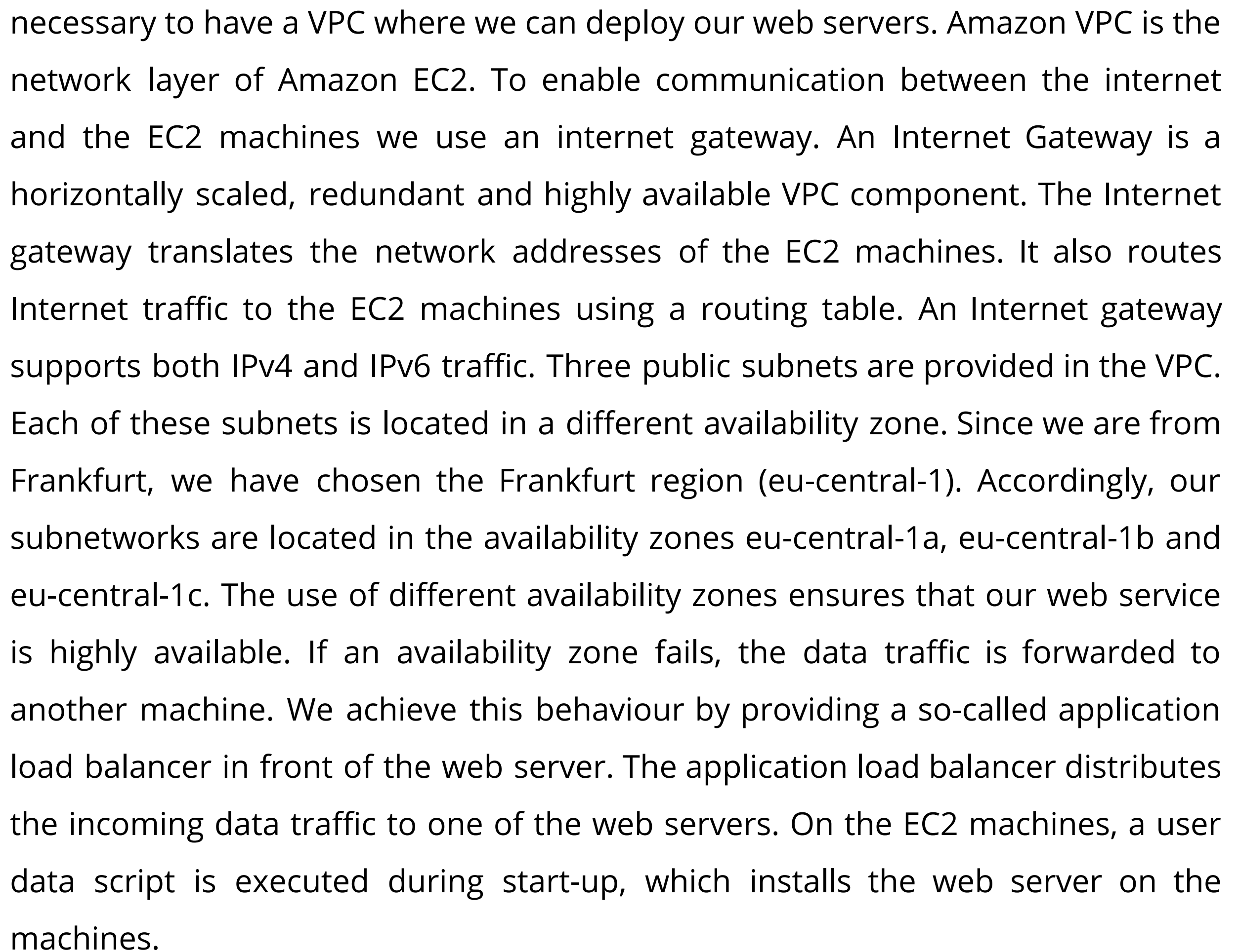


Figure 3: Architecture

The project is being realised using the AWS Cloud. The infrastructure consists of several services that are described here (see the figure 3 above). First of all, it is



necessary to have a VPC where we can deploy our web servers. Amazon VPC is the network layer of Amazon EC2. To enable communication between the internet and the EC2 machines we use an internet gateway. An Internet Gateway is a horizontally scaled, redundant and highly available VPC component. The Internet gateway translates the network addresses of the EC2 machines. It also routes Internet traffic to the EC2 machines using a routing table. An Internet gateway supports both IPv4 and IPv6 traffic. Three public subnets are provided in the VPC. Each of these subnets is located in a different availability zone. Since we are from Frankfurt, we have chosen the Frankfurt region (eu-central-1). Accordingly, our subnetworks are located in the availability zones eu-central-1a, eu-central-1b and eu-central-1c. The use of different availability zones ensures that our web service is highly available. If an availability zone fails, the data traffic is forwarded to another machine. We achieve this behaviour by providing a so-called application load balancer in front of the web server. The application load balancer distributes the incoming data traffic to one of the web servers. On the EC2 machines, a user data script is executed during start-up, which installs the web server on the machines.

3. Creating AWS-Account / Set-up

We created an AWS account at *http://aws.amazon.com* to use the web service AWS offers that we need. For log in we need only root credentials: e-mail, password and account ID (see the figure 4 below). These credentials allow unrestricted access to all resources in the account. With AWS Identity and Access Management (IAM - feature of the AWS account) we can manage access to AWS services and

resources. With IAM we can create and manage AWS users and groups and also define permissions, for example to deny the access to AWS resources.

Figure 4: Anmeldung

Create IAM-User

Because it is not possible to restrict the permissions for root users, we deleted root access keys contained in the account after creating the account and that the so-called IAM users be created in AWS Identity and Access Management (IAM) instead. Logging in as an IAM user is always done with an account ID, the corresponding IAM username and a password. The root account is needed only for a few tasks such as:

- Change the root user details
- Changing payment options

- Changing the support plan
- Retrieval of billing information
- Transfer one Route 53 domain to another AWS.

For log in IAM user as a root user on the login screen, click on the corresponding link below the login button.

The root account can also be protected by multi-factor authentication (hardware or software token) if desired. In our case it is not necessary. AWS also supports different types of federation with other authentication solutions (OpenID, SAML, AD-Federation, etc.). The corresponding settings can be found by clicking on the account name in the upper right corner of the AWS management console (see the figure 5 below).

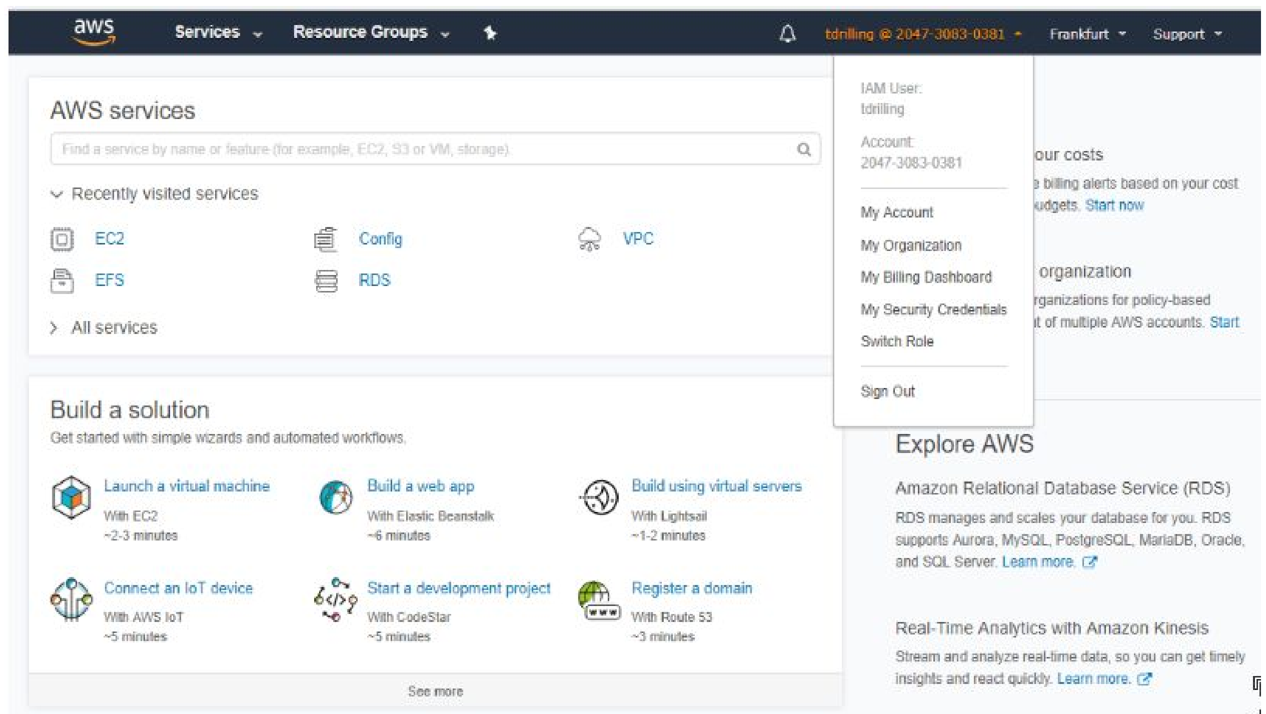


Figure 5: AWS Management Console

If you are logged in as an IAM user, the login name <iam-user@account-id> appears in the upper right hand corner. In the case of a root user, the written

username with first and last name is written out. As a root user you can then edit various account settings with a click on my account.

AWS- Account set-up

For the correct execution of the Terraform script, it is necessary that the administrator responsible for the infrastructure has programmatic access to the AWS console. Assuming that the administrator has the necessary access, the profile must be stored on the machine executing the script. The following steps help to set up an AWS profile:

1. A credentials file must be created on the computer:

a. For Linux and Mac, this file must be located under the path **~/. aws/config**. b. For Windows this file must be located under the path **%USERPROFILE%\. aws\config .**

2. Here is an example of the content of credentials-file:

```
1.  [default]
2.  aws_access_key_id=AWS_ACCESS_KEY_ID
3.  aws_secret_access_key=AWS_SECRET_ACCESS_KEY
```

3. A config file must be created on the computer:

a. The config file is located under the same path as the credentials file for both operating systems.

4. Each profile can specify different credentials, possibly from different IAM users and can also specify different AWS regions and output formats.

Here is an example of the contents of a config file:

```
1.  [default]
2.  region=IAM-Benutzer-Region
3.  output=json oder text
```

4. Terraform installation and configuration

I. Terraform installation for Windows

After Terraform has been downloaded and installed, the successful installation can be verified by entering terraform in the terminal. If the terraform command is not detected, it is very likely that the Terraform binary file was not stored on the **PATH**. The path is the system variable with which your operating system (Windows) searches the required executable files via the command line or the terminal window. To add the file to the system path, you must first go to the system menu and then open the **Advanced System Settings** window (see the figure 6 below).

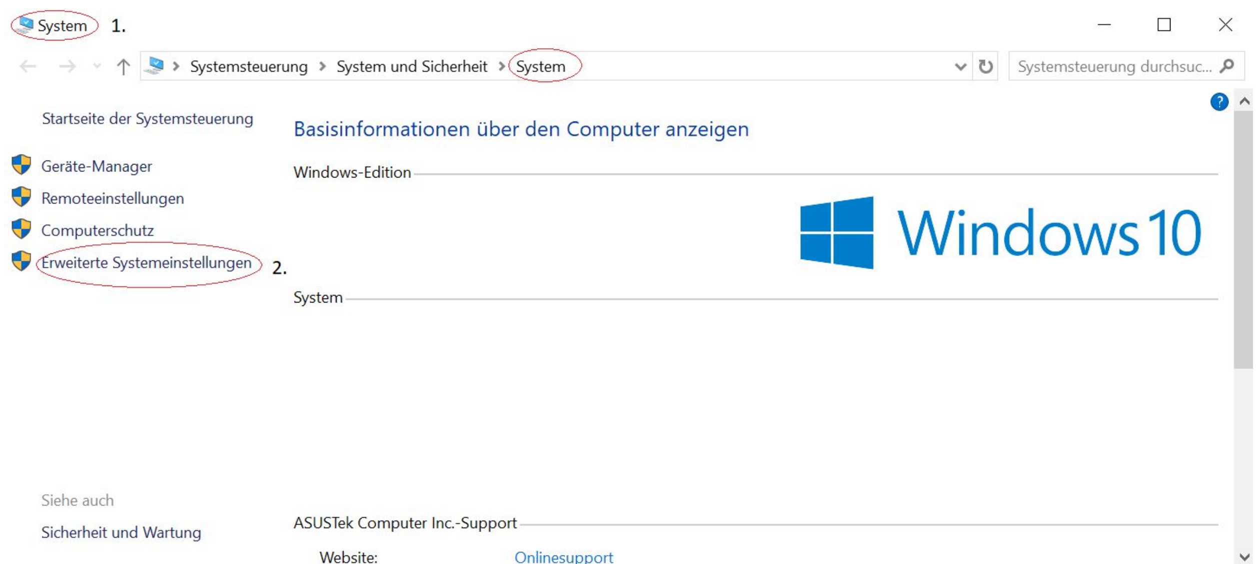


Figure 6: System Settings

Subsequently click on the environment variables (see the figure 7 below).

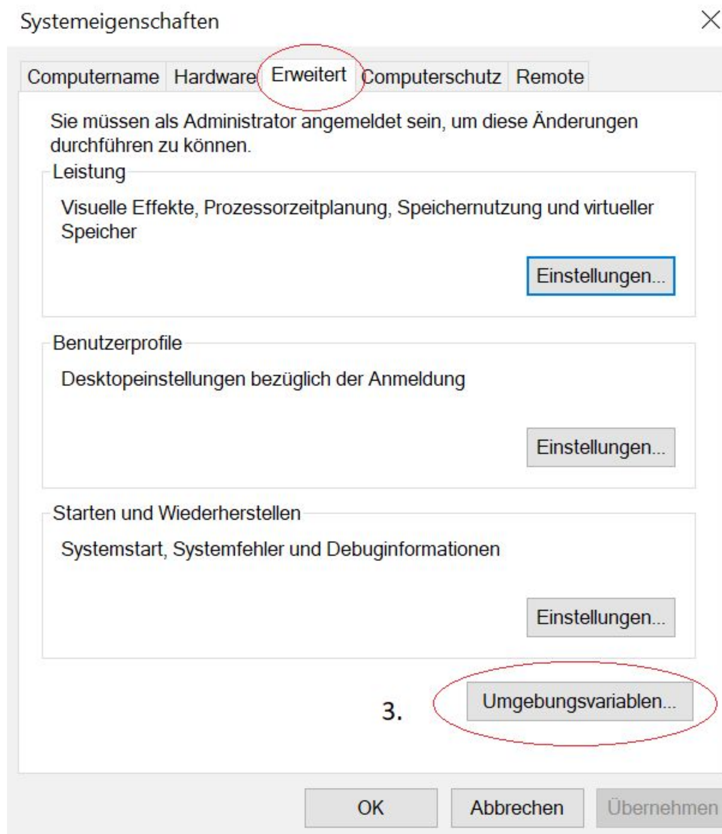


Figure 7: Environment Variables

Now in the lower window System Variables Path can be clicked

(see the figure 8 below).

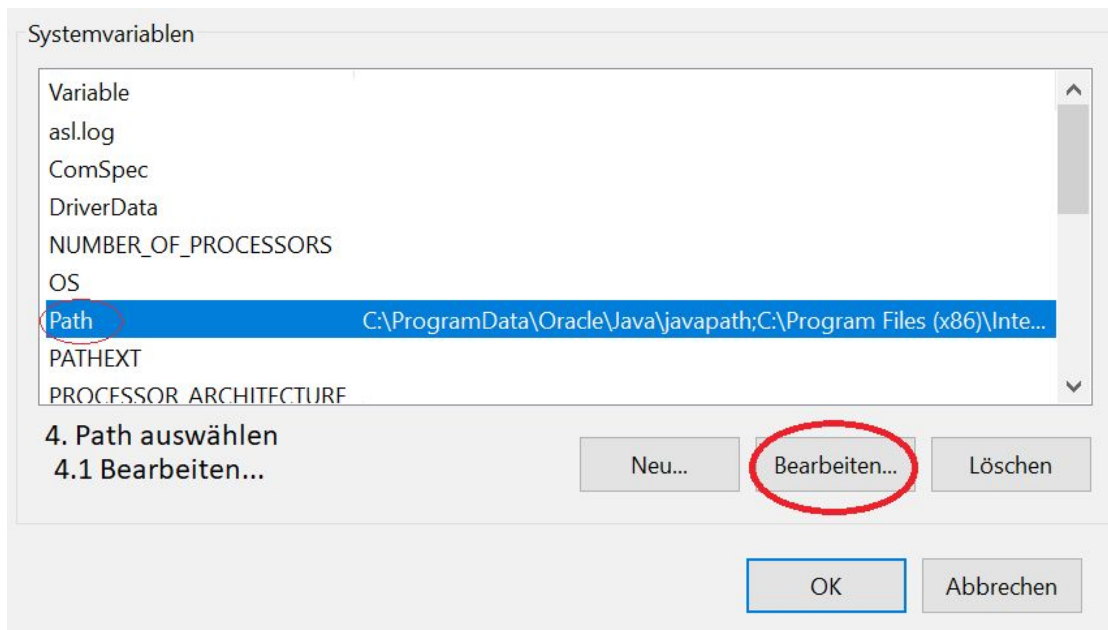


Figure 8: System Variables

Now a new environment variable must be added. There should be the Terraform path, for example ***C:\Users\PfadZuTerraform*** (see the figure 9 below).

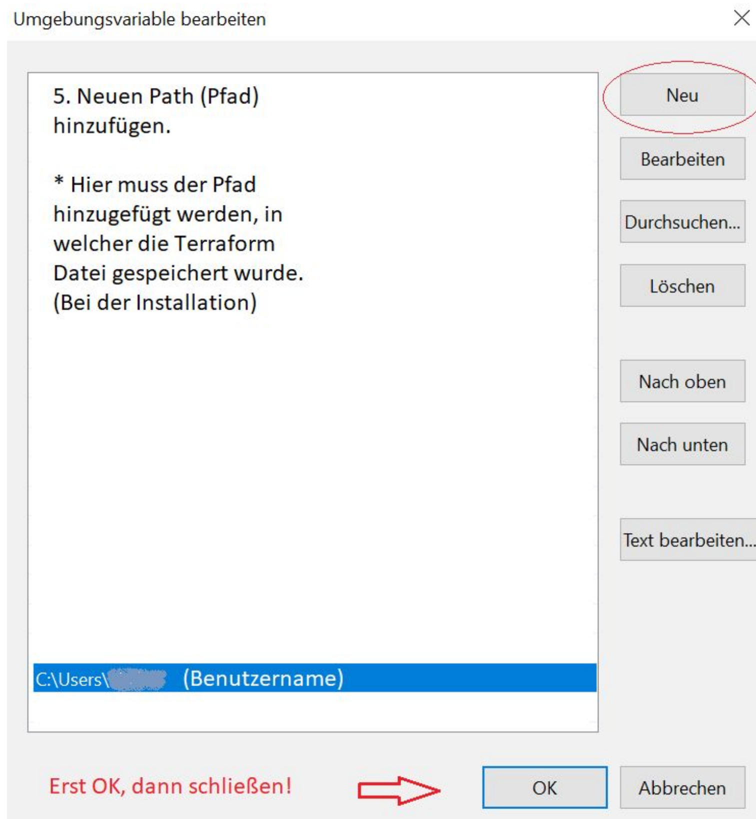


Figure 9: Terraform Path

After that click on OK and you can try again to confirm the installation with terraform in the terminal. It should work from any path.

For Linux

Download Terraform for the respective operating system and run the installation. After that, it is necessary to confirm installation with the terraform command in the terminal.

The following output should be shown:

Usage: terraform [-version] [-help] <command> [args] …

II. Configuration

If Terraform is used for an AWS deployment, it has to be configured. Therefore a main.tf file is created

```
1. provider "aws" {
2.     version = "~> 2.27"
3.     region  = var.region
4.   }
```

Now Terraform will apply all operations on behalf of the AWS named account. Sometimes it's worth it to create a separate user for Terraform only.

Terraform uses its own scripting language called HashiCorp Configuration Language (HCL), which allows to apply some additional programming logic like variables, mapping and conditions inside your template. It's possible to pass variables to the template and get certain outputs.

For better granulation it is possible to split big templates into several parts. It's possible with Terraform modules.

There is a way to integrate Terraform with configuration management tools, using Provisioners, for example to run some commands on newly created EC2 instances.

By default Terraform stores the state and the history of all the state changes locally. It's possible and recommended to organize a remote state storage for better teamwork, e.g via S3 or file hosting service.

Resources are described in the "resource" section. That's how an resource will look for creating a VPC:

```
1.  provider "aws" {
2.     version = "~> 2.27"
3.     region  = var.region
4.    }
5.  resource "aws_vpc" "vpc" {
```

```
6.    cidr_block         = var.vpc_cidre_block
7.    enable_dns_hostnames = true
8.    enable_dns_support = true
9.    tags = {
10.       Name = var.vpc_name
11.   }
12. }
```

Where aws_vpc is a resource type, vpc - resource name and everything inside of curly braces are arguments or resource properties.

Terraform execution

To run the Terraform script, you must navigate to the **. /terraform/ directory** in the terminal. Then the **terraform init** command must be entered and executed with an enter. This command is used to initialize the working directory including the configuration files. Once the working directory has been initialized, it is no longer necessary to execute this command again. The **terraform validate** command is then executed to check the written code for syntax errors.

The optional **terraform refresh** command is used to coordinate the actual state. In the next step, the terraform plan command is executed. The ***terraform plan*** command is for creating an execution plan. Terraform performs a refresh and then specifies the actions that will achieve the desired state determined in the configuration file.

Next, ***terraform apply*** provides the creation of our infrastructure. If everything worked good, the output should be: ***Apply complete! Resources created***. can be read in the terminal. Another output indicates an error in the script. In this case, the output in the terminal must identify and fix the error by the script execute (debugging).

The infrastructure is destroyed by executing the command ***terraform destroy*** in the terminal in the project directory.

5. Development of Terraform Script

In this block the provider is specified.

```
1. provider "aws" {
2.     version = "~> 2.27"
3.     region  = var.region
4.   }
```

Creating of AWS VPC, separate dedicated VPC.

```
1. resource "aws_vpc" "vpc" {
2.    cidr_block = var.vpc_cidre_block
3.    enable_dns_hostnames = true
4.    enable_dns_support = true
5.
6.    tags = {
7.        Name = var.vpc_name
8.    }
9. }
```

Deploying in every availability zone in the region one public subnet. Additionally to that we are tagging the subnets. The cidr blocks for the subnets are coming from a list which is stored in variables.tf. The subnets are added to our vpc.

```
1. resource aws_subnet "vpc_public_subnet"{
2.
3.    count = length(var.public_subnets)
4.    vpc_id = aws_vpc.vpc.id
5.    cidr_block = element(var.public_subnets, count.index)
6.    availability_zone = element(var.availability_zone, count.index)
7.    map_public_ip_on_launch = true
8.
9.    tags = {
```

```
10.         Name   =  "${var.vpc_name}-public-subnet-${element(var.availability_zone,
   count.index)}"
11.   }
12. }
```

For the communication with the internet we are deploying an internet gateway. The gateway is added to our vpc. Since the creation of the internet gateway depends on that the vpc already exists we have added here a depends_on argument to be sure that the internet gateway is created after the vpc and we are not running in any dependency issues.

```
1. resource "aws_internet_gateway" "vpc_gateway" {
2.   vpc_id = aws_vpc.vpc.id
3.   tags = {
4.     Name = var.vpc_gateway
5.   }
6.   depends_on = [aws_vpc.vpc]
7. }
```

We add the route to the Internet to the Main Route table of the VPC. The Main Route table is created by default. This is in turn assigned to the Internet Gateway.

```
1. resource "aws_route" "internet_access" {
2.   route_table_id = aws_vpc.vpc.main_route_table_id
3.   destination_cidr_block = "0.0.0.0/0"
4.   gateway_id = aws_internet_gateway.vpc_gateway.id
5. }
```

```
1. resource "aws_route_table_association" "public" {
2.   count = length(var.public_subnets)
3.   subnet_id = element(aws_subnet.vpc_public_subnet.*.id, count.index)
4.   route_table_id  = aws_vpc.vpc.main_route_table_id
5. }
```

The logic for creating the AWS Key Pair for the ec2 machines now follows. This key is encrypted with the RSA algorithm. For this, the 4096 bits variant is used to ensure more security. The private key is stored locally. To enable a connection with the ec2 machines, the permissions on the private key are adjusted. This adjustment is necessary because otherwise AWS complains that the private key is too open. Therefore, only the owner of the key is allowed to have read access. The key is not essential for the infrastructure. It is only created to enable an SSH connection from the developers' machines to try out configurations without having to tear down the entire infrastructure.

```
1. resource "tls_private_key" "key_pair" {
2.   algorithm = "RSA"
3.   rsa_bits  = 4096
4. }
```

```
1. resource "aws_key_pair" "generated_key" {
2.   key_name = var.key_name
3.   public_key = tls_private_key.key_pair.public_key_openssh
4. }
```

```
1. resource "local_file" "private_key" {
2.   content  = tls_private_key.key_pair.private_key_pem
3.   filename = "./test.pem"
4. }
```

```
1. resource "null_resource" "private_key_permissions" {
2.   depends_on = [local_file.private_key]
3.
4.   provisioner "local-exec" {
5.     command     = "chmod 400 ./test.pem"
6.     interpreter = ["bash", "-c"]
7.     on_failure  = continue
8.   }
9. }
```

A web service is to be hosted on the ec2 machines. This web service is to be accessible via the internet and provided on different machines for the purpose of high availability. For this purpose, the use of an application load balancer makes sense. why the decision was made in favour of the application load balancer instead of the network balancer is quite clear. we want to balance the data traffic on the http port. the network balancer balances the data traffic on the network layer. The load balancer is assigned a security group, which is defined later in the code.

```
1. resource "aws_lb" "webserver_alb" {
2.   name = var.webserver_alb_name
3.   internal = false
4.   load_balancer_type = "application"
5.   security_groups = [aws_security_group.security_group_alb.id]
6.   subnets = aws_subnet.vpc_public_subnet.*.id
7.   tags = {
8.     Name = var.webserver_alb_name
9.   }
10. }
```

A target group is created for the load balancer. In our case, we want to target the ec2 machines on the subnets.

```
1. resource "aws_lb_target_group" "webserver_alb_tg" {
2.   name = var.webserver_tg_name
3.   target_type = "instance"
4.   port = 80
5.   protocol = "HTTP"
6.   vpc_id = aws_vpc.vpc.id
7.
8.   tags = {
9.   Name = var.webserver_tg_name
10.   }
11. }
```

a forward rule is created for the alb. This tells the load balancer which machines it should forward to when it is addressed.

```
1.  resource "aws_lb_listener" "webserver_alb_listener" {
2.    load_balancer_arn = aws_lb.webserver_alb.arn
3.    port = "80"
4.    protocol = "HTTP"
5.    default_action {
6.      type = "forward"
7.      target_group_arn = aws_lb_target_group.webserver_alb_tg.arn
8.    }
9.  }
```

Here the machines are added to the target group

```
1.  resource "aws_lb_target_group_attachment" "webserver_alb_attachment" {
2.    count = 3
3.    target_group_arn = aws_lb_target_group.webserver_alb_tg.arn
4.    target_id = element(aws_instance.webserver.*.id, count.index)
5.    port= 80
6.     }
```

This code snippet is used to create the web server. Three ec2 machines are created on each subnet in our vpc. the machines receive the operating system from a data block. Data blocks are used to search resources for information. We use the data block defined in data.tf to search aws for the new ubuntu version and assign it to our ec2 machine. For cost reasons, we have chosen t2.nano as the instance type. By providing a user data script, we bootstrap the ec2 machines with the desired ec2 machines. In other words, we tell them which steps have to be carried out when booting up. In addition, we assign the previously created key to the ec2 machine so that we can use this key to establish an ssh connection to the machines. the ec2 machines are assigned a public ip address. This means that they can be reached from the internet. We also assign the size of the hard disk to the machine.

```
1.  resource "aws_instance" "webserver" {
2.      count = var.server_count
```

```
3.      ami = data.aws_ami.ami.image_id
4.      instance_type = var.instance_type
5.      user_data = templatefile("./installWebServer.sh.tpl", {})
6.      key_name = var.key_name
7.      subnet_id = element(aws_subnet.vpc_public_subnet.*.id, count.index)
8.      associate_public_ip_address = true
9.      availability_zone = element(var.availability_zone, count.index)
10.
11. vpc_security_group_ids =    [aws_security_group.security_group_webserver.id]
12.
13.     depends_on = [aws_subnet.vpc_public_subnet]
```

```
1.    root_block_device {
2.        volume_size = var.volume_size
3.    }
4.
5.     tags = {
6.    Name = var.server_name
7.      }
8.         }
```

The various security groups for the web servers and the application load balancer now follow. We have created a separate security group for each service. The security group for the web server allows access from outside (ingress) to ports 80 (http) and 22 (ssh) while the security group for the load balancer only allows access to port 80. In both security groups, however, all communication is allowed from the outside (egress).

```
1.    resource "aws_security_group" "security_group_webserver" {
2.    name = var.vpc_security_group_webserver
3.    vpc_id = aws_vpc.vpc.id
4.
5.     tags = {
6.     Name = var.vpc_security_group_webserver
7.     }
8.    }
```

```
1.    resource "aws_security_group" "security_group_alb" {
2.    name = var.vpc_security_group_alb
3.    vpc_id = aws_vpc.vpc.id
4.
5.    tags = {
6.    Name = var.vpc_security_group_alb
7.     }
8.     }
```

```
1.    resource "aws_security_group_rule" "allow_http1" {
2.    type = "ingress"
3.    description = "HTTP Rule for WebServer"
4.    from_port = 80
5.    to_port = 80
6.    protocol = "tcp"
7.    cidr_blocks = ["0.0.0.0/0"]
8.
9.    security_group_id = aws_security_group.security_group_alb.id
10.    }
```

```
1.    resource "aws_security_group_rule" "allow_outbound_traffic1"{
2.    type = "egress"
3.    description = "Allow outbound traffic to the internet"
4.    from_port = 0
5.    to_port = 0
6.    protocol = -1
7.    cidr_blocks = ["0.0.0.0/0"]
8.
9.  security_group_id = aws_security_group.security_group_alb.id}
10.
11.   resource "aws_security_group_rule" "allow_ssh" {
12.   type = "ingress"
13.   description = "Allow ssh traffic for Administration    reasons"
14.   from_port = 22
15.   to_port = 22
16.   protocol = "tcp"
17.   cidr_block = ["0.0.0.0/0"]
```

```
18.
19.   security_group_id = aws_security_group.security_group_webserver.id
20. }
```

```
1.  resource "aws_security_group_rule" "allow_http" {
2.    type                 = "ingress"
3.    description          = "HTTP Rule for WebServer"
4.    from_port            = 80
5.    to_port              = 80
6.    protocol             = "tcp"
7.    cidr_blocks          = ["0.0.0.0/0"]
8.
9.  security_group_id = aws_security_group.security_group_webserver.id
10. }
```

```
1.  resource "aws_security_group_rule" "allow_outbound_traffic" {
2.    type                 = "egress"
3.    description          = "Allow outbound traffic to the internet"
4.    from_port            = 0
5.    to_port              = 0
6.    protocol             = -1
7.    cidr_blocks          = ["0.0.0.0/0"]
8.    security_group_id = aws_security_group.security_group_webserver.id
9.  }
```

6. Ansible installation and configuration

This Chapter describes the Ansible Installation on Ubuntu/Windows and the configuration.

I. Ansible installation on Ubuntu

With Ansible it is possible to control automated a lot of different systems from the one location. With Ansible it is possible to build a simple architecture without special software that must be installed on nodes. This tool uses SSH to carry out the automation tasks and YAML files for specifying provisioning details.

Before using Ansible as a manager of infrastructure, it is necessary to install the Ansible software on the computer that will work as the Ansible control node.

```
1. $ sudo apt-add-repository ppa:ansible/ansible
```

As the next, press Enter and accept the PPA addition.

Next, it is need to refresh the system's package index for the available packages in the new PPA:

```
1. $ sudo apt update
```

After the update, it is necessary to install the Ansible software :

```
1. $ sudo apt install ansible
```

The Ansible control node has all required software to orchestrate the hosts [2].

II. Ansible installation on Windows

First of all we run the following script in CMD to set-up WinRM for Ansible:

```
1.     powershell.exe -ExecutionPolicy ByPass -File
2.     "windows-host-setup.ps1"
```

Here you can see the script "windows-host-setup.ps1":

```
1.     $ url = "https://raw.githubusercontent.com/ansible/
2.     ansible/devel/examples/scripts/ConfigureRemotingForAnsible.ps1"
3.     $ file = "$env:temp\ConfigureRemotingForAnsible.ps1"
4.     (New-Object -TypeName System.Net.WebClient).DownloadFile
5.     ($url,$file)
6.     powershell.exe -ExecutionPolicy ByPass -File $file
```

As next, you need to execute Ansible playbook on Windows. Then to run Ansible control node [15].

III. Ansible configuration

The settings in Ansible are regulable via a configuration file *ansible.cfg*. Paths where configuration file is located, is to find in reference documentation. By installing Ansible from a package manager, the latest *ansible.cfg* file should be present in /etc/ansibleas a *.rpmnew* file in the case of updates.

Ansible enables configuration of settings with environment variables. The determined environment variables will override the setting, which will be loaded from the configuration file.

Not all configuration options are present in the command line not all config options existing , only the most common. Due to the configuration file and the environment the settings will be in the command line override. The full list of options existing in ansible and ansible-playbook[4].

In the default case, ansible's configuration file is located at /etc/ansible/ansible.cfg. Normally, the default configurations are sufficient to get you started using Ansible. The list of all configs existing in the control node, yo must to use the command ansible-config [3]:

```
1.  $ ansible-config list
```

You can see the output at the figure 10 below.

```
ACTION_WARNINGS:
  default: true
  description:
  - By default Ansible will issue a warning when received from a task action (m>
    or action plugin)
  - These warnings can be silenced by adjusting this setting to False.
  env:
  - name: ANSIBLE_ACTION_WARNINGS
  ini:
  - key: action_warnings
    section: defaults
  name: Toggle action warnings
  type: boolean
  version_added: '2.5'
AGNOSTIC_BECOME_PROMPT:
  default: true
  description: Display an agnostic become prompt instead of displaying a prompt>
    the command line supplied become method
  env:
  - name: ANSIBLE_AGNOSTIC_BECOME_PROMPT
  ini:
  - key: agnostic_become_prompt
    section: privilege_escalation
:
```

Figure 10: Ansible Config

7.Development of Ansible Script

First of all we need to create three EC2 Instances using Ansible. Before we go into the playbooks part, we need to check and update a few environment set-up.

Environment Setup for Ansible to work with AWS EC2 module.

The modules of Ansible are written in python. That's why for working with AWS modules it is necessary to install prerequisite elements on Ansible machine:

- boto
- botocore
- boto3
- python version 2.6.

Boto is one of the Amazon SDK and boto3 is the newest version of boto.

Then you need to execute the Python in the terminal and type **import boto** and **import boto3**. Prerequisite for installing both boto and boto 3 is that you have already **pip3**.

Below is the code for creating EC2 instances and for getting the list of our AWS Cloud account. The code is divided in two blocks (group of tasks) :

- The first block is for the instances information
- The second block is to create the instances [1].

The first block is for the instances information

Here is the code for deploying the simple AWS Infrastructure present.

```
1.  - name: Deploy simple AWS Infrastructure
```

We use a local host where the actual script is launched from.

```
1.  hosts: localhost vars:
```

We have AWS Region eu-central 1.

```
1.  aws_region: eu-central-1
2.  vpc_name: Test_VPC_AP
3.  igw_name: Test_IGW_AP
4.  image_name: ami-0e1ce3e0deb8896d2
5.  tasks:
```

Creating of AWS VPC, separate dedicated VPC.

```
1.  - name: Create AWS VPC
2.      ec2_vpc_net:
3.      name: "{{ vpc_name }}"
```

As next, will the range of IPs for VPC specify.

```
1.  cidr_block: 10.10.0.0/16
2.  region: "{{ aws_region }}"
3.  tenancy: default
4.  register: VPC_AP.
```

Creating of AWS VPC IGW for accessing the VPC from outside (internet).

```
1.  -name: Create AWS VPC IGW
2.  ec2_vpc_igw:
3.  vpc_id: "{{ VPC_AP.vpc.id }}"
4.  region: "{{ aws_region }}"
5.  state: present
6.  register: IGW_AP
```

Creating of Subnet1 AWS VPC.

```
1.  -name: Create AWS VPC Subnet1
2.   ec2_vpc_subnet:
3.   state: present
4.   vpc_id: "{{ VPC_AP.vpc.id }}"
```

Specifying the range of IPs for Subnet 1. CIDR - classless Inter-Domain routing.

```
1.  cidr: 10.10.16.0/20
2.  region: "{{ aws_region }}"
```

Specifying the availability zone for Subnet 1.

```
1.  az: eu-central-1a
2.  register: Test_subnet_1_AP
```

Creating of Subnet2 AWS VPC.

```
1.   -name: Create AWS VPC Subnet2
2.   ec2_vpc_subnet:
3.   state: present
4.   vpc_id: "{{ VPC_AP.vpc.id }}"
5.   cidr: 10.10.32.0/20
6.   region: "{{ aws_region }}"
```

Specify availability zone for Subnet 2.

```
1.     az: eu-central-1b
2.     register: Test_subnet_2_AP
```

Creating of Subnet 3 AWS VPC.

```
1.  -name: Create AWS VPC Subnet3
```

```
2.  ec2_vpc_subnet:
3.  state: present
4.  vpc_id: "{{ VPC_AP.vpc.id }}"
5.  cidr: 10.10.48.0/20
6.  region: "{{ aws_region }}"
```

Specify availability zone for Subnet 1.

```
1.  az: eu-central-1c
2.  register: Test_subnet_3_AP
```

Creating a Subnet AWS VPC Routing Table.

```
1.  -name: Create AWS VPC Subnet Routing Table
2.  ec2_vpc_route_table:
3.  vpc_id: "{{ VPC_AP.vpc.id }}"
4.  region: "{{ aws_region }}"
```

Here is the list of subnets which elb has to have the access to.

```
1.  subnets:
2.  -"{{ Test_subnet_1_AP.subnet.id }}"
3.   -"{{ Test_subnet_2_AP.subnet.id }}"
4.  -"{{ Test_subnet_3_AP.subnet.id }}"
5.  routes:
6.  -dest: 0.0.0.0/0
7.  gateway_id: "{{ IGW_AP.gateway_id }}"
8.  register: route_table_AP
```

Creating of AWS Security Group.

```
1.  -name: Create AWS Security Group
2.  ec2_group:
3.  name: Test_SG_AP
```

Creating of AWS Security Group für AP.

```
1.  description: AWS Security Group for AP
2.  vpc_id: "{{ VPC_AP.vpc.id }}"
3.  region: "{{ aws_region }}"
```

Rules for inbound traffic on ports 22, 80,443.

```
1.  rules:
2.  rules_egress:
3.  -proto: tcp
4.  from_port: 80
5.  to_port: 80
6.  cidr_ip: 0.0.0.0/0
7.  -proto: tcp
8.  from_port: 22
9.  to_port: 22
10. cidr_ip: 0.0.0.0/0
11. -proto: tcp
12. from_port: 443
13. to_port: 443
```

Inbound traffic on ports 22. 80,443 will be allowed.

```
1.  ruesc: Allow inbound traffic on ports 22 (SSH), 80 (HTTP),   443 (HTTPS)
2.  rulesle_d_egress:
3.   - proto: tcp
4.  from_port: 80
5.  to_port: 80
```

```
6.  cidr_ip: 0.0.0.0/0
7.  rule_desc: Allow outbound traffic
8.  on ports 22 (SSH), 80 (HTTP), 443 (HTTPS)
9.  register: sec_group
```

The second block is to create the instances

Below we can see how instances are created. Here will EC2 instance created for Subnet 1:

```
1.  - name: Create EC2 instance for Subnet
2.  ec2_instance:
3.  name: seite1
4.  region: "{{ aws_region }}"
5.  instance_type: t2.micro
6.  user_data: "{{ lookup('file', 'user_data.sh') }}"
7.  image_id: "{{ image_name }}"
8.  wait: yes
```

Here you can see as well that all of our created instances belong to the same security group.

```
1.  security_group: Test_SG_AP
2.  vpc_subnet_id: "{{ Test_subnet_1_AP.subnet.id }}"
3.  network:
4.  assign_public_ip: true
5.  register: inst1
```

Here will be an EC2 instance created for Subnet 2.

```
1.  -name: Create EC2 instance for Subnet 2
2.  ec2_instance:
3.  name: seite2
4.  region: "{{ aws_region }}"
```

```
5.  instance_type: t2.micro
6.  user_data: "{{ lookup('file', 'user_data.sh') }}"
7.  image_id: "{{ image_name }}"
8.  wait: yes
9.  security_group: Test_SG_AP
10. vpc_subnet_id: "{{ Test_subnet_2_AP.subnet.id }}"
11. network:
12. assign_public_ip: true
13. register: inst2
```

EC2 instance for Subnet 3 is created.

```
1.  -name: Create EC2 instance for Subnet 3
2.  ec2_instance:
3.  name: seite3
4.  region: "{{ aws_region }}"
5.  instance_type: t2.micro
6.  user_data: "{{ lookup('file', 'user_data.sh') }}"
7.  image_id: "{{image_name}}"
8.  wait: yes
9.  security_group: Test_SG_AP
10. vpc_subnet_id: "{{ Test_subnet_3_AP.subnet.id }}"
11. network:
12. assign_public_ip: true
13. register: inst3
```

Create a target group see health check in amazon, in target group we specify instances, where the traffic is routed to.

```
1.  -name:    Create    a    target    group    with    a    default    health    check
    elb_target_group:
2.  name: testTG
3.  health_check_path: /
4.  protocol: http
5.  port: 80
6.  vpc_id: "{{ VPC_AP.vpc.id }}"
7.  targets:
```

```
8.  - Id: "{{ inst1.instances[0].instance_id }}"
9.  Port: 80
10. - Id: "{{ inst2.instances[0].instance_id }}"
11.   Port: 80
12.  - Id: "{{ inst3.instances[0].instance_id }}"
13.  Port: 80
14.  state: present
```

Creating Application ELB, the elb target group that was created earlier, is bound to the elb.

```
1.  -name: Create Application ELB
2.  elb_application_lb:
3.  name: TestAppELBAP
4.  region: "{{ aws_region }}"
5.  security_groups:
6.  - "{{ sec_group.group_id }}"
7.   subnets:
```

For elb, the list of subnets which elb has to have the access to.

```
1.  - "{{ Test_subnet_1_AP.subnet.id }}"
2.  - "{{ Test_subnet_2_AP.subnet.id }}"
3.  - "{{ Test_subnet_3_AP.subnet.id }}"
4.  listeners:
5.  - Protocol: HTTP
6.  Port: 80
7.  DefaultActions:
8.   - Type: forward
9.  TargetGroupName: testTG
10. state: present
```

8. Comparing Ansible and Terraform

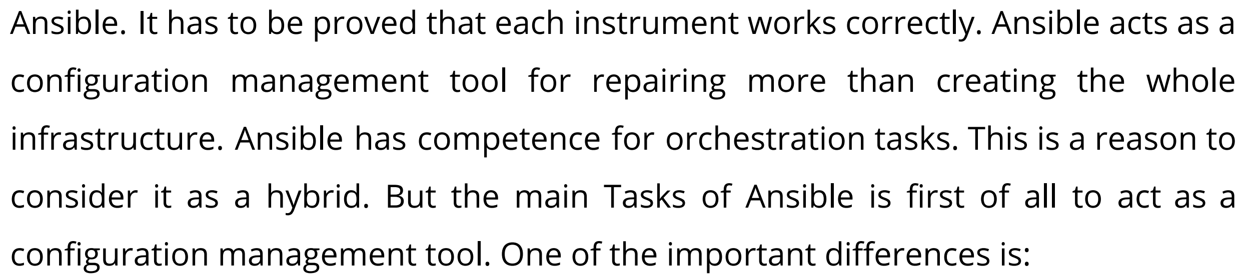
Nowadays Ansible and Terraform are famed in the DevOps landscape. They both tools are well- known for their unambiguous advantages in creating infrastructure as code. The tools offer infrastructure as a Code that is very helpful in deploying repeatable environments with complex requirements. They both hasInfrastructure as a Code. It means that terraform and ansible are automate: configuring, provisioning and managing the infrastructure.

Let’s have a look and compare the main differences between Ansible and Terraform.The first signifikant difference between it is orchestration and configuration management.

- Ansible is a configuration management tool vs. Terraform is a tool for orchestration.

Of course there are many similarities between the functionalities of configuration management and orchestration, but it is very important to know the differences in details. Because a distinct understanding of the differences between these tools is very helpful in choosing the right applications. For better understanding of the functionality of Ansible and Terraform by, it is important to know some details. Terraform focuses mainly on the final objectives and Terraform always underlines on maintaining a special state of the environment. Terraform stores the state of the environment and in this way provides a better foundation for recovery. So, Terraform can provide the resource automatically upon running it again. It is the perfect tool for maintaining steady-state environments.

Ansible is a configuration management tool. One of the tasks of a configuration management tool is equivalent to repairing instruments in an orchestration.All components of an environment in working condition can be maintained with



Ansible. It has to be proved that each instrument works correctly. Ansible acts as a configuration management tool for repairing more than creating the whole infrastructure. Ansible has competence for orchestration tasks. This is a reason to consider it as a hybrid. But the main Tasks of Ansible is first of all to act as a configuration management tool. One of the important differences is:

- Ansible uses procedural and Terraform uses declarative language

The way their work gives us a good opportunity to compare the tools. Let's have a look into differences with Terraform based on procedural or declarative processes. Many DevOps tools can be categorized into procedural and declarative categories.

The procedural category indicates applications that required the same steps presented in the code. For example, by increasing/ scaling down EC2 instances, it is necessary to determine the number of instances.

Declarative tools offer an exact impression of the requirements. For example, If you need 3 EC2 instances for scaling down your environment, you must determine the exact number. Terraform follows the declarative language. With Terraform the environment must be determined particularly. “Terraform Apply” can adjust any changes in the environment.

Ansible is in fact a hybrid of procedural and declarative. It is possible to execute ad-hoc commands for procedural configuration. It also exists the opportunity of using different Ansible modules that can carry out tke declarative configuration [6].

The summary of differences in Ansible vs Terraform you can see below.

Main Differences between Ansible and Terraform

Point of Difference	Terraform	Ansible
Type	Mainly infrastructure provisioning tool/ Orchestration tool	Mainly configuration toolInstall/Update software on that infrastructure
Support	Only partial Support for packaging and templating. Terraform offers direct access to HashiCorp's support	Complete Support for packaging and templating. THe tool provides 2 levels of professional support for the enterprise version
Ease set-up and usage	Tool is simpler to use and to set-up. THe users can use a proxy server for the installation	It is easy to install and use. The tool has a master without agents (agentless), running on the client machines. Ansible uses YAML syntax (Python)
Lifecycle management	Lifecycle management	No Lifecycle management
Infrastructure	Provides support for immutable infrastructure	Provides support for mutable infrastructure
Availability	Not Applicable	The tool has a secondary node in case an active node not function
Modules	The modules offer for users an abstract away of any reusable parts. The parts must be configured	Ansible Galaxy available, it consists of a repository or library

	only once and can be used everywhere	
GUI	Only 1/3 parts of GUIs are available. For example, Codeherent's Terraform GUI	GUI is presented as a command-line tool. The enterprise version provides a UI, but that does not fulfil the expectations
Language	Uses declarative language	Uses procedural language
Tool	Relatively new	More mature

Difference between Ansible and Terraform for AWS

Both Terraform and Ansible process AWS management quite variously.

Terraform with AWS

Terraform is the perfect tool for users who do not have a lot of practical knowledge to manage AWS. Nevertheless it is not so easy to act with Terraform. There are a few benefits of using Terraform with AWS:

- Terraform is open-source with it common benefits of utilizing open-source software
- In the event of an error the dependent resources will isolate. Non-dependent resources can be created, updated and also destroyed.
- Preview changes before the applying is possible.
- JSON support and a user-friendly syntax

Ansible with AWS

Ansible provides good support for AWS. With the help of Ansible playbooks it is possible to utilize also the complex AWS environments.Users can deploy them many times and scale out to thousands of instances across different regions. Ansible has about 100 modules that support AWS capabilities. For example, Simple Storage Service (S3), Security Token Service, Relational Database Service, Virtual Private Cloud, Security Groups, Route53, Identity Access Manager, etc. It also provides about 1300 additional modules for managing various requirements of a user's Windows, Linux, UNIX.

Here are some benefits of using Ansible with AWS:

- Ansible Tower's cloud inventory synchronization helps to find out which AWS instances register
- Security in automation with its set of role-based access controls
- Control inventory by keeping track of deployed infrastructure via the life cycles. Consequently it shows us that security policies execute correctly.
- The same simple playbook language manages infrastructure and deploys applications to different infrastructures easily [7].

9. Summary

It is not so easy to find the answer for the questions: ” What to choose - Terraform or Ansible”. Mainly it depends of course on the requirements. These tools have many similarities and differences. Which tool to choose? What tool is better? From the practice side it is recommendable to use Terraform for orchestration and Ansible configuration management. Also many technology companies search for the best solution among these two tools for automating apps and for creating their IT Infrastructure. But there is no perfect tool, it depends on what it is used for.

The main task of Terraform is orchestration. Terraform has all necessary updates that are perfect for orchestration. The command “Terraform Plan” can provide more helpful information as the “Ansible-dry-run” command. In turn, Ansible is the perfect tool for configuration management. But take into consideration, the Ansible orchestration tasks are limited. In comparison to Terraform Ansible is more tricky in use. If you do not have experience in Ansible, first of all you must learn to automate the deployment, configuration and management of the infrastructure. You need much more time for the learning Ansible, because the documentation of Ansible has only minimal basic information.

References

[1] AWS ALB Target Group shows unhealthy instances in a custom VPC. URL: https://stackoverflow.com/questions/65610989/aws-alb-target-group-shows-unhealthy-instances-in-a-custom-vpc. [Accessed on 2021-01-07].

[2] How to install and configure Ansible on Ubuntu. URL:https://www.digitalocean.com/community/tutorials/how-to-install-and-configure-ansible-on-ubuntu-18-04 [Accessed on 2021-01-10].

[3] Getting started with Ansible - Basic Installation and setup. URL: https://www.linode.com/docs/guides/getting-started-with-ansible/[Accessed on 2021-01-09].

[4] Ansible latest installation Guide. URL: https://docs.ansible.com/ansible/latest/installation_guide/intro_configuration.html#id5 [Accessed on 2021-01-10].

[5] Using Terraform to Manage AWS Programmable Infrastructures. URL: https://aws.amazon.com/de/blogs/apn/using-terraform-to-manage-aws-programmable-infrastructures/ [Accessed on 2021-01-10].

[6] Ansible vs Terraform: Understanding the Differences. URL:https://www.whizlabs.com/blog/ansible-vs-terraform/ [Accessed on 2021-01-10].

[7] Ansible vs Terraform vs Puppet:Which to Choose? URL: https://phoenixnap.com/blog/ansible-vs-terraform-vs-puppet. [Accessed on 2021-01-11].

[8] Was ist Infrastructure as Code (IaC)? https://www.cloudcomputing-insider.de/was-ist-infrastructure-as-code-iac-a-917671/[Accessed on 2021-01-13].

[9] 15 Infrastructure as Code tools you can use to automate your deployments URL:https://www.thorntech.com/2018/04/15-infrastructure-as-code-tools/ [Accessed on 2021-01-30].

[10] Introduction to Terraform. URL: https://www.terraform.io/intro/index.html [Accessed on 2021-01-30].

[11] Command: plan URL: https://www.terraform.io/docs/cli/commands/plan.html [Accessed on 2021-01-30].

[12] DevOps101 — First Steps on Terraform: Terraform + OpenStack + Ansible URL: https://medium.com/hackernoon/terraform-openstack-ansible-d680ea466e22 [Accessed on 2021-01-30].

[13] What Is Ansible? – Configuration Management And Automation With Ansible URL: https://www.edureka.co/blog/what-is-ansible/ [Accessed on 2021-01-30].

[14] OVERVIEW How Ansible Works URL: https://www.ansible.com/overview/how-ansible-works [Accessed on 2021-01-30].

[15] Getting started with Ansible: local automation of Windows 10 and Ubuntu 20.04 workstations. URL: https://levelup.gitconnected.com/getting-started-with-ansible-local-automation-of-windows-10-and-ubuntu-20-04-workstations-ffd03d7dc92 [Accessed on 2021-01-30].

[16] Ansible or Terraform - a shot answer URL: https://medium.com/faun/ansible-or-terraform-a-short-answer-90a9fd8bb0aa [Accessed on 2021-02-04].