Report DevOps class - Efrei 2022

Arradi Naoufal, Bienfait Meo, Bouchakour Younes, Sergent Pierre-Louis, Tadjer Badr

https://github.com/PLsergent/devops-efrei (https://github.com/PLsergent/devops-efrei)

Table of contents

Pre requisite

- Install Ansible (`./commands_install.md`)
- Install Terraform (`./commands_install.md`)
- Install Packer (`./commands_install.md`)

If you're doing the following Terraform on your own machine export AWS credential keys:

```
export AWS_ACCESS_KEY_ID=AKIAIOSFODNN7EXAMPLE
export AWS_SECRET_ACCESS_KEY=wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
```

Ansible

Machines:

- *ToolsCICD*: 172.16.1.41
- *RemoteServer*: 172.16.1.201

Introduction

Ping

```
ubuntu@ip-172-16-1-41:~$
ansible all -m ping -i ubuntu@172.16.1.201, --private-key ~/.ssh/key.pem
ubuntu@172.16.1.201 | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python3"
    },
    "changed": false,
    "ping": "pong"
}
```

First playbook

```
---
- hosts: all
  tasks:
        - name: test connection
          ping:
```

hosts

```
[remote_server]
172.16.1.201 ansible_ssh_private_key_file=~/.ssh/key.pem
```

Launch playbook

```
ubuntu@ip-172-16-1-41:~/TP_CICD/web_ami$
ansible-playbook -i hosts playbook.yml

PLAY [all]
********************************************************************************
*****************************

TASK [Gathering Facts]
********************************************************************************
*****************
ok: [172.16.1.201]

TASK [test connection]
********************************************************************************
*****************
ok: [172.16.1.201]

PLAY RECAP
********************************************************************************
*****************************
172.16.1.201               : ok=2    changed=0    unreachable=0    failed=0
skipped=0    rescued=0    ignored=0
```

Website

Configuration & playbook

hosts

```
[remote_server]
172.16.1.201 ansible_ssh_private_key_file=~/.ssh/key.pem
```

We use a template file for the apache virtual host configuration.

virtualhost.conf

```
<VirtualHost *:{{ http_port }}>
    DocumentRoot /var/www/{{ domain }}
    ErrorLog ${APACHE_LOG_DIR}/error.log
    CustomLog ${APACHE_LOG_DIR}/access.log combined
</VirtualHost>
```

playbook-website.yml

```
---
- hosts: remote_server
  become: yes
  vars:
    http_port: 8080
    domain: html

  tasks:
    - name: install packages
      apt:
        name: ['apache2', 'git']
        update_cache: yes
        state: latest

    - name: enabled mod_rewrite
      apache2_module: name=rewrite state=present
      notify:
        - restart apache2

    - name: apache2 listen on port {{ http_port }}
      lineinfile: dest=/etc/apache2/ports.conf regexp="^Listen 80" line="Listen
{{ http_port }}" state=present
      notify:
        - restart apache2

    - name: apache2 virtual host
      template: src=virtualhost.conf dest=/etc/apache2/sites-available/000-
default.conf

    - name: Remove default /var/www/html/
      file:
        path: /var/www/html
        state: absent

    - name: clone basic html template
      git: repo=https://github.com/cloudacademy/static-website-example.git
dest=/var/www/html update=no
      notify:
        - restart apache2

  handlers:
    - name: restart apache2
      service: name=apache2 state=restarted
```

Launch playbook-website.yml

```
ubuntu@ip-172-16-1-41:~/TP_CICD/web_ami$ ansible-playbook -i hosts playbook-
website.yml

PLAY [remote_server]
********************************************************************************
```

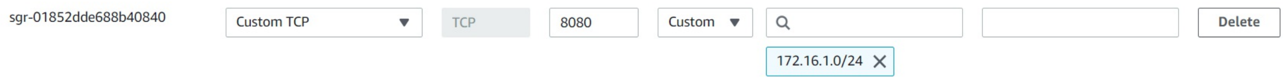
```
*******************************************************

TASK [Gathering Facts]
********************************************************************************
*****************************************************
ok: [172.16.1.201]

TASK [install packages]
********************************************************************************
****************************************************
ok: [172.16.1.201]

TASK [enabled mod_rewrite]
********************************************************************************
*************************************************
ok: [172.16.1.201]

TASK [apache2 listen on port 8080]
********************************************************************************
*****************************************
ok: [172.16.1.201]

TASK [apache2 virtual host]
********************************************************************************
************************************************
ok: [172.16.1.201]

TASK [Remove default /var/www/html/]
********************************************************************************
***************************************
changed: [172.16.1.201]

TASK [clone basic html template]
********************************************************************************
*******************************************
changed: [172.16.1.201]

RUNNING HANDLER [restart apache2]
********************************************************************************
******************************************
changed: [172.16.1.201]

PLAY RECAP
********************************************************************************
*****************************************************************
172.16.1.201               : ok=8    changed=3    unreachable=0    failed=0
skipped=0    rescued=0    ignored=0
```

Check result

Check from the remote server:

```
ubuntu@ip-172-16-1-201:~$
systemctl status apache2
● apache2.service - The Apache HTTP Server
   Loaded: loaded (/lib/systemd/system/apache2.service; enabled; vendor preset:
enabled)
  Drop-In: /lib/systemd/system/apache2.service.d
           └─apache2-systemd.conf
   Active: active (running) since Thu 2022-03-17 16:16:17 UTC; 1min 41s ago
  Process: 31124 ExecStop=/usr/sbin/apachectl stop (code=exited,
status=0/SUCCESS)
  Process: 31129 ExecStart=/usr/sbin/apachectl start (code=exited,
status=0/SUCCESS)
 Main PID: 31143 (apache2)
    Tasks: 55 (limit: 1140)
   CGroup: /system.slice/apache2.service
           ├─31143 /usr/sbin/apache2 -k start
           ├─31145 /usr/sbin/apache2 -k start
           └─31146 /usr/sbin/apache2 -k start
```

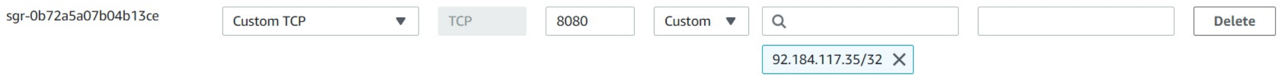
Check from the tool_cicd server (after allowing the port 8080 from the subnet 172.16.1.0/24):



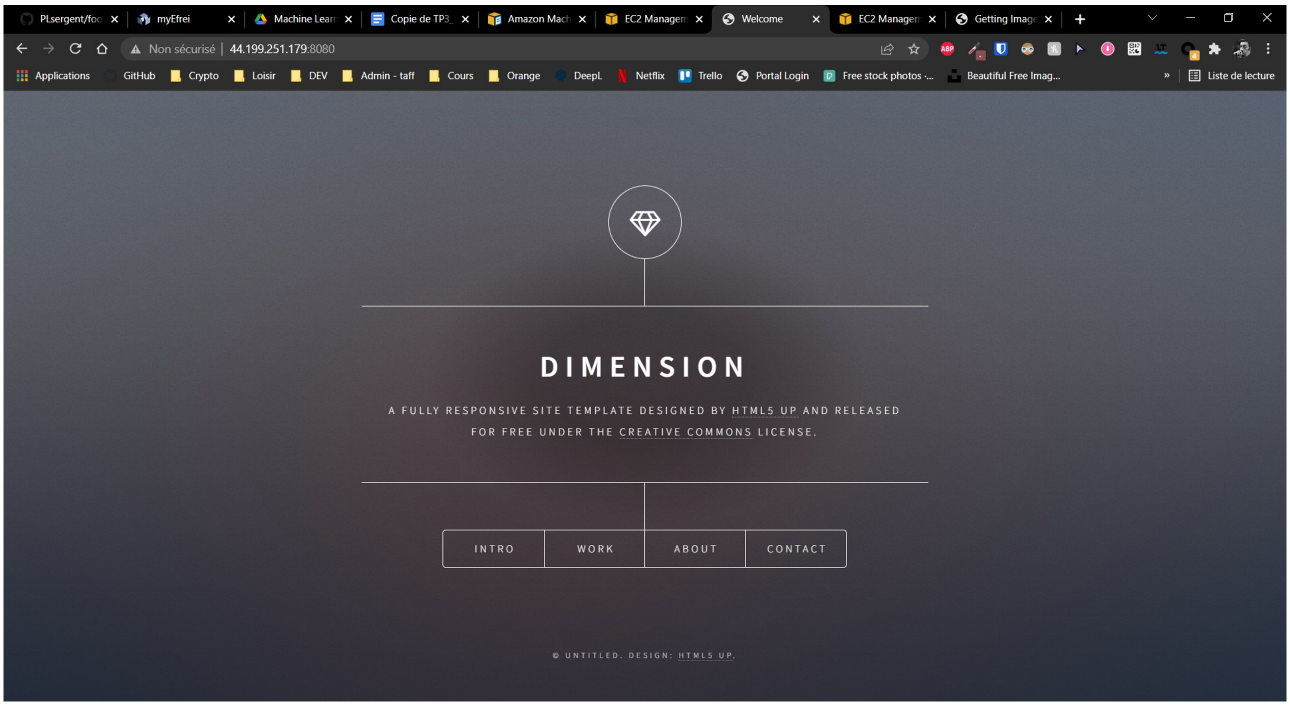
sgr-01852dde688b40840	Custom TCP	TCP	8080	Custom	172.16.1.0/24		Delete

```
lynx http://172.16.1.201:8080
...
curl http://172.16.1.201:8080
```

Check from a public machine (after allowing port 8080 from my ip):



sgr-0b72a5a07b04b13ce	Custom TCP	TCP	8080	Custom	92.184.117.35/32		Delete



DIMENSION

A FULLY RESPONSIVE SITE TEMPLATE DESIGNED BY HTML5 UP AND RELEASED FOR FREE UNDER THE CREATIVE COMMONS LICENSE.

© UNTITLED. DESIGN: HTML5 UP.

Terraform

Packer

Using the file `./web_ami/buildAMI.json` and change those parameters specific to our configuration:

- region: "eu-west-1",
- ssh_username: "ubuntu",
- base_ami: "ami-07d8796a2b0f8d29c", => used the build the temporary ec2 instance and is the base for the new AMI
- instance_type: "t2.micro",
- subnet_id: "subnet-0f00dc50bf4e0723b",
- temporary_security_group_source_cidrs": "X.X.X.X/32"

Use the command `packer validate` to validate the json file.

```
devops-efrei/terraform/web_ami on  main
❯ packer validate buildAMI.json
Template validated successfully.
```

```
❯ packer build buildAMI.json
```

Packer is then going to create a temporary ec2 instance based on the base_ami id given.
It will then use the given ansible playbook (`./web_ami/play.yml`) to install setup the instance.
Finally it will use this configured instance to create a new AMI.

We now have our AMI for the web app.

Terraform files

Variables

File: `variables.tf`

We used three variables:

- region: eu-west-1
- ami_id: the id of the AMI we created previously
- base_ami_id: the AMI id of a typical Ubuntu micro ec2 instance

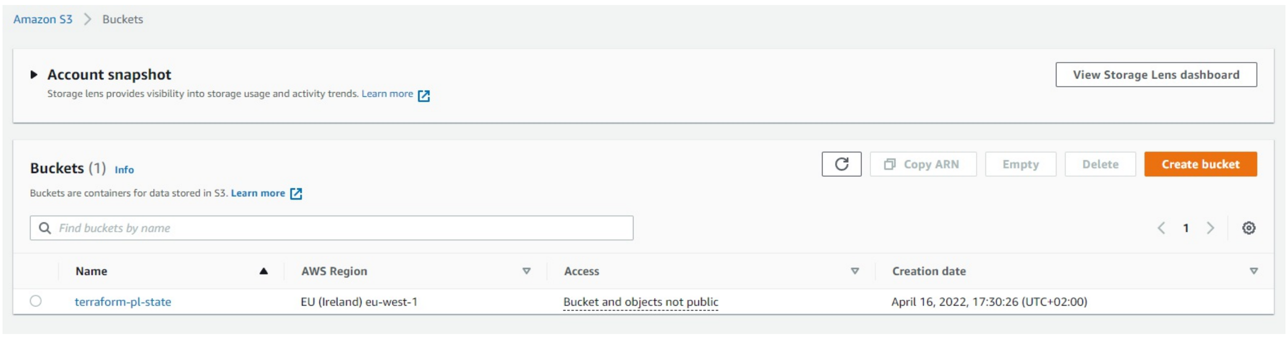
Since we only needed the default value of thoses variables we didn't need to define them in a file `*.tfvars`.

```
variable "region" {
    description = "The AWS region to use"
    default = "eu-west-1"
}

variable "ami_id" {
    description = "value of the AMI id"
    default = "ami-0b9c9dccd6f4d9f1b"
}

variable "base_ami_id" {
    description = "value of the base AMI id"
    default = "ami-07d8796a2b0f8d29c"
}
```

Provider and Terraform state

File: `provider.tf`

In order to store the terraform state we created manually an s3 bucket.



Storage lens provides visibility into storage usage and activity trends. Learn more

Buckets (1) Info

Buckets are containers for data stored in S3. Learn more

Name	AWS Region	Access	Creation date
terraform-pl-state	EU (Ireland) eu-west-1	Bucket and objects not public	April 16, 2022, 17:30:26 (UTC+02:00)

We can then reference this bucket in our `provider.tf` file.

```
terraform {
  backend "s3" {
    bucket         = "terraform-pl-state"
    key            = "tf_infra/terraform.tfstate"
    region         = "eu-west-1"
  }
}
```

The Terraform state represents the state of the infrastructure at a given time. Although it's not mandatory, it's a good idea to store the state in a bucket to facilitate coworking on the same project. Otherwise it will be stored locally in a file named `terraform.tfstate`.

Before each `terraform plan` and `terraform apply` Terraform will look into the state and then compare our code to the current infrastructure. If there is a difference, Terraform will plan to create or destroy the necessary resources. It will also `refresh` the state by looking at the actual infrastructure and make the necessary change in the state.

Even if it's possible, it's better to not change the infrastructure manually because it might create conflict between the actual infrastructure and our code. Moreover, we should not change the state manually only if there is a major issue, it's even better to use the commands `terraform import` and `terraform state rm` in that situation.

The only **provider** we need here is AWS so we import it like so:

```
provider "aws" {
    region  = "eu-west-1"
}
```

The provider will allow us to use the AWS API to create, destroy, and manage the infrastructure.

VPC, subnets & NAT

File: `vpc.tf`

This file contains the definition of the VPC (Virtual Private Cloud), subnets and NAT.

In order to facilitate the coding of the vpc resources we used an Amazon module.

```
module "vpc" {
  source = "terraform-aws-modules/vpc/aws"

  name = "vpc"
  cidr = "10.0.0.0/16"

  azs              = ["${var.region}a", "${var.region}b", "${var.region}c"]
  public_subnets   = ["10.0.1.0/24", "10.0.2.0/24", "10.0.3.0/24"]
  private_subnets  = ["10.0.4.0/24", "10.0.5.0/24", "10.0.6.0/24"]

  enable_nat_gateway = false
  enable_vpn_gateway = false

  tags = {
    Terraform = "true"
  }
}
```

A module is a collection of resources that are used together. It helps us to abstract the creation of all the necessary resources for a vpc. It also helps to avoid the repetition of the same code.

The module will create a VPC with the differents subnets provided in the `public_subnets` and `private_subnets` variables. The availability zones will be the ones provided in the azs variable.

It will also create the internet gateway and the route tables.

For the NAT instance we used the module ec2-instance from AWS, with a security group that we created.

```
// Create a Nat instance in the VPC
module "ec2_instance" {
  source  = "terraform-aws-modules/ec2-instance/aws"
  version = "~> 3.0"

  name = "nat-instance"

  ami                    = var.base_ami_id
  instance_type          = "t2.micro"
  key_name               = "aws-pl"
  monitoring             = true
  vpc_security_group_ids = [aws_security_group.nat_instance_sg.id]
  subnet_id              = module.vpc.public_subnets[0]

  tags = {
    Terraform   = "true"
  }

  depends_on = [
    aws_security_group.nat_instance_sg // should be created before the instance
  ]
}

// Create a security group for the NAT instance
resource "aws_security_group" "nat_instance_sg" {
  name        = "nat-instance-sg"
  description = "Allow instances in the private subnets to access internet."
  vpc_id      = module.vpc.vpc_id

  ingress {
    description      = "Private subnets instances acces nat instance"
    from_port        = 0
    to_port          = 0
    protocol         = "-1"
    cidr_blocks      = module.vpc.private_subnets
  }

  egress {
    description      = "Allow internet access from nat instance"
    from_port        = 0
    to_port          = 0
    protocol         = "-1"
    cidr_blocks      = ["0.0.0.0/0"]
  }

  tags = {
    Name = "nat-instance-sg"
  }

  depends_on = [
    module.vpc
  ]
}
```

We used the base AMI to create the NAT instance. Also the security group will, as described, allow the instance to access: the internet and the instances in the private subnets.

We can use the value of the previously created resources: `subnet_id = module.vpc.public_subnets[0]`, this way we can create the NAT instance in the first subnet of the VPC. Therefor if the subnets cidr would change, we wouldn't have to change mnually the value of the `subnet_id` variable.

Autoscaling group

File: `asg.tf`

An autoscaling group is a collection of instances that can be scaled up and down. Once initialized the ash will then scale up and down depending on the load of the instances, meaning new instances will be created automaticaly. It's also possible to define a minimum and maximum number of instances.

In order to know when to scale up and down we can define an autoscaling policy or initialize a cloud watch alarm.

```
resource "aws_placement_group" "asg_placement_group" {
  name = "asg-placement-group"
  strategy = "cluster"
}

// Lauch configuration for the ASG
resource "aws_launch_configuration" "ec2_lc" {
  name_prefix   = "ec2-lc-"
  image_id      = var.ami_id
  instance_type = "t2.micro"

  lifecycle {
    create_before_destroy = true
  }
}

// Auto scaling group resource
resource "aws_autoscaling_group" "asg" {
    name = "asg-ec2-instance"
    launch_configuration = aws_launch_configuration.ec2_lc.id
    min_size = 1
    max_size = 4
    vpc_zone_identifier = module.vpc.private_subnets
    load_balancers = [ aws_elb.elb.id ]
    placement_group = aws_placement_group.asg_placement_group.name

    depends_on = [
      aws_launch_configuration.ec2_lc,
      aws_placement_group.asg_placement_group
    ]
}

// Auto scaling group policy
resource "aws_autoscaling_policy" "example" {
  name = "asg-ec2-instance-policy"
  adjustment_type = "ChangeInCapacity"
  scaling_adjustment = 1
  autoscaling_group_name = aws_autoscaling_group.asg.name
  cooldown = 300

  target_tracking_configuration {
    predefined_metric_specification {
      predefined_metric_type = "ASGAverageCPUUtilization"
    }

    target_value = 40.0
  }
}
```

Prior the creation of the ASG we need to create the placement group. Then we defined a lauch configuration which is simply the configuration of the ec2 instances to launch when the asg scales up. For the `image_id` we used the id of the AMI we created earlier.

Then in the asg itself we can define the minimum and maximum number of instances to launch as well as the subnets in which the instances will be launched. The launch configuration and the placement group are also passed to the resource parameters.

Finally, we create the policy that will be used to scale up and down the ASG with the following parameters:

- **adjustment_type**: `ChangeInCapacity`

Meaning that when the asg scales up or down, it will change the number of instances by the value of the `scaling_adjustment` parameter.

- **scaling_adjustment**: 1

Meaning that when the asg scales up or down, it will change the number of instances by 1.

- **cooldown**: 300

Meaning that the asg will not scale up or down for 300 seconds after a scaling.

- **target_tracking_configuration**:
    - **predefined_metric_specification**:
        - **predefined_metric_type**: `ASGAverageCPUUtilization`
        - **target_value**: 40.0

Meaning that the asg will scale up when the average CPU utilization is above 40%.

Elastic Load Balancer

File: `elb.tf`

The elastic load balancer is a simple load balancer that can be used to balance the incoming traffic.

```
resource "aws_elb" "elb" {
    name = "elb-ec2-instance"
    subnets = module.vpc.public_subnets
    security_groups = [aws_security_group.elb_sg.id]
    listener {
        instance_port = 80
        instance_protocol = "http"
        lb_port = 80
        lb_protocol = "http"
    }
    listener {
        instance_port      = 80
        instance_protocol  = "http"
        lb_port            = 443
        lb_protocol        = "https"
        ssl_certificate_id = "arn:aws:iam::123456789012:server-
certificate/certName"
    }
    tags = {
        Name = "elb-ec2-instance"
    }

    depends_on = [
      aws_security_group.elb_sg // should be created before the instance
    ]
}

// Create a security group for the ELB
resource "aws_security_group" "elb_sg" {
    name        = "elb-sg"
    description = "Allow instances in the private subnets to access internet."
    vpc_id      = module.vpc.vpc_id

    egress {
        description      = "Allow private subnets access from elb"
        from_port        = 0
        to_port          = 0
        protocol         = "-1"
        cidr_blocks      = module.vpc.private_subnets
    }

    ingress {
        description      = "Allow internet access to elb"
        from_port        = 80
        to_port          = 80
        protocol         = "-1"
        cidr_blocks      = ["0.0.0.0/0"]
    }

    ingress {
        description      = "Allow internet access to elb"
        from_port        = 443
        to_port          = 443
        protocol         = "-1"
        cidr_blocks      = ["0.0.0.0/0"]
    }
}
```

The elb will be located in the public subnets as specified on the infrastructure diagram.

We are setting two listeners on the elb. The first one will be listening on port 80 and will be used for the HTTP protocol. The second one will be listening on port 443 and will be used for the HTTPS protocol.

For the security group we are allowing access from the internet and to the private subnets. It's logical since the elb need to redirect the traffic to the instances located in the private subnets.

If we go back to the asg we can also add the elb to the load balancers list so the asg is aware that we are using a load balancer.

Outputs

File: `output.tf`

This file contains the outputs of our terraform code, meaning that after the apply it will print the provided values.

Even though it's optional it's useful to have the outputs of the terraform code in order to debug the terraform code or directly have informations about the deployed resources.

```
output "elb_dns_name" {
    value = aws_elb.elb.dns_name
}

output "nat_instance_id" {
    value = module.ec2_instance.id
}
```

Here we kept it simple and just print the DNS name of the elb and the id of the nat instance.

Commands

- `terraform init`: initialize a working directory containing Terraform configuration files
- `terraform plan`: plan the changes to be made to your infrastructure
- `terraform apply`: apply the planned changes to your infrastructure

Our Terraform plan to create 27 new resources. In order to not spend money we're not going to apply the plan.