CONSTRUIRE UNE INFRASTRUCTURE AWS HAUTEMENT DISPONIBLE

Introduction

Jusqu'ici, nous avons fait le point sur beaucoup de notions de Terraform. Dans ce chapitre un peu spécial, nous réaliserons un TP en utilisant les éléments vus ensemble dans les chapitres précédents.

Comme promis dans le chapitre sur la réalisation de notre première infrastructure AWS depuis Terraform, le but de ce TP est de **réaliser depuis Terraform une infrastructure AWS hautement disponible** pour une application en php communiquant avec une base de données relationnelle.

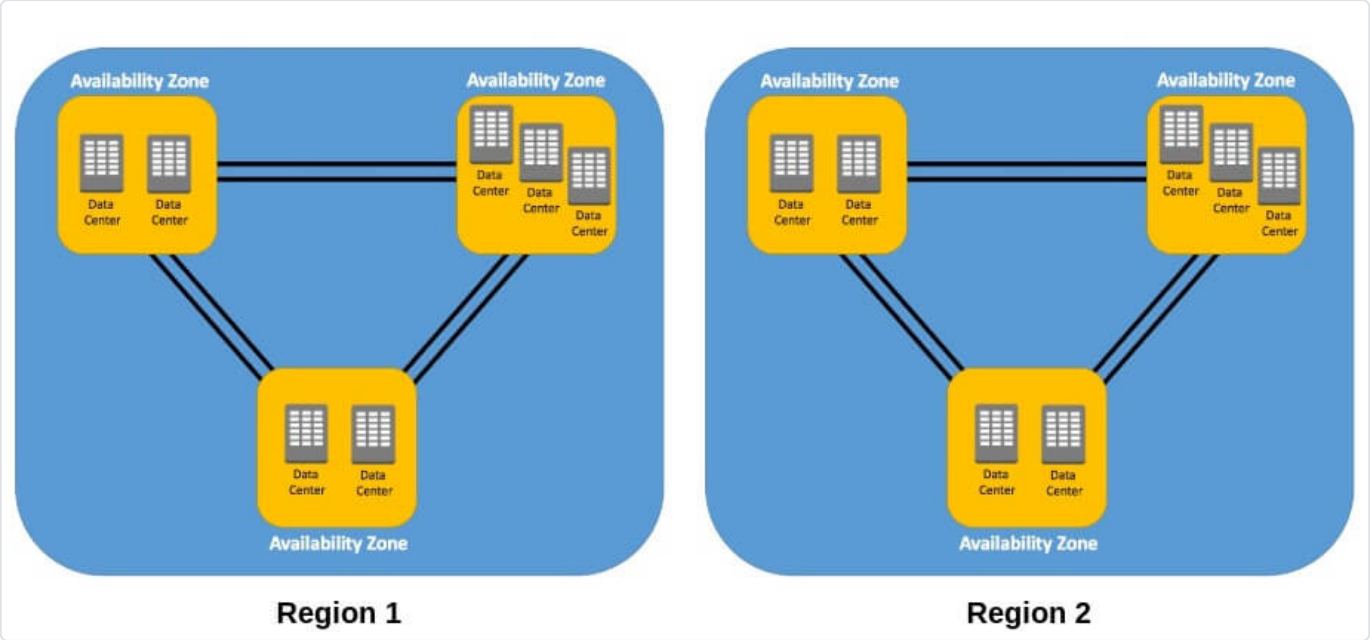
Qu'est-ce que la haute disponibilité ?

Il convient tout d'abord d'**expliquer le terme "haute disponibilité"**. Les systèmes dits hautement disponibles ("High Availability (HA)" en anglais) sont **fiables** dans le sens où ils continuent de fonctionner même en cas de défaillance de composants critiques. Ils sont également **résilients**, ce qui signifie qu'ils sont capables de **gérer les pannes sans interruption de service ou perte de données** de manière transparente pour les utilisateurs finaux.

La haute disponibilité dans AWS

La géographie

Amazon Web Services (AWS) propose une large gamme de services que nous pouvons exploiter pour implémenter la haute disponibilité à toute application Web que nous déployons sur le Cloud AWS. Avant de commencer le déploiement de notre application, voyons voir déjà **comment AWS organise son infrastructure à travers le monde** depuis un schéma :



- AWS est réparti dans le monde entier sur plusieurs sites géographiques nommés régions.
- Dans chaque région, il existe plusieurs emplacements isolés appelés zones de disponibilité ("Availability Zones (AZ)" en anglais). Une zone de disponibilité peut être considérée comme un ensemble de data-centers distincts dans une région.

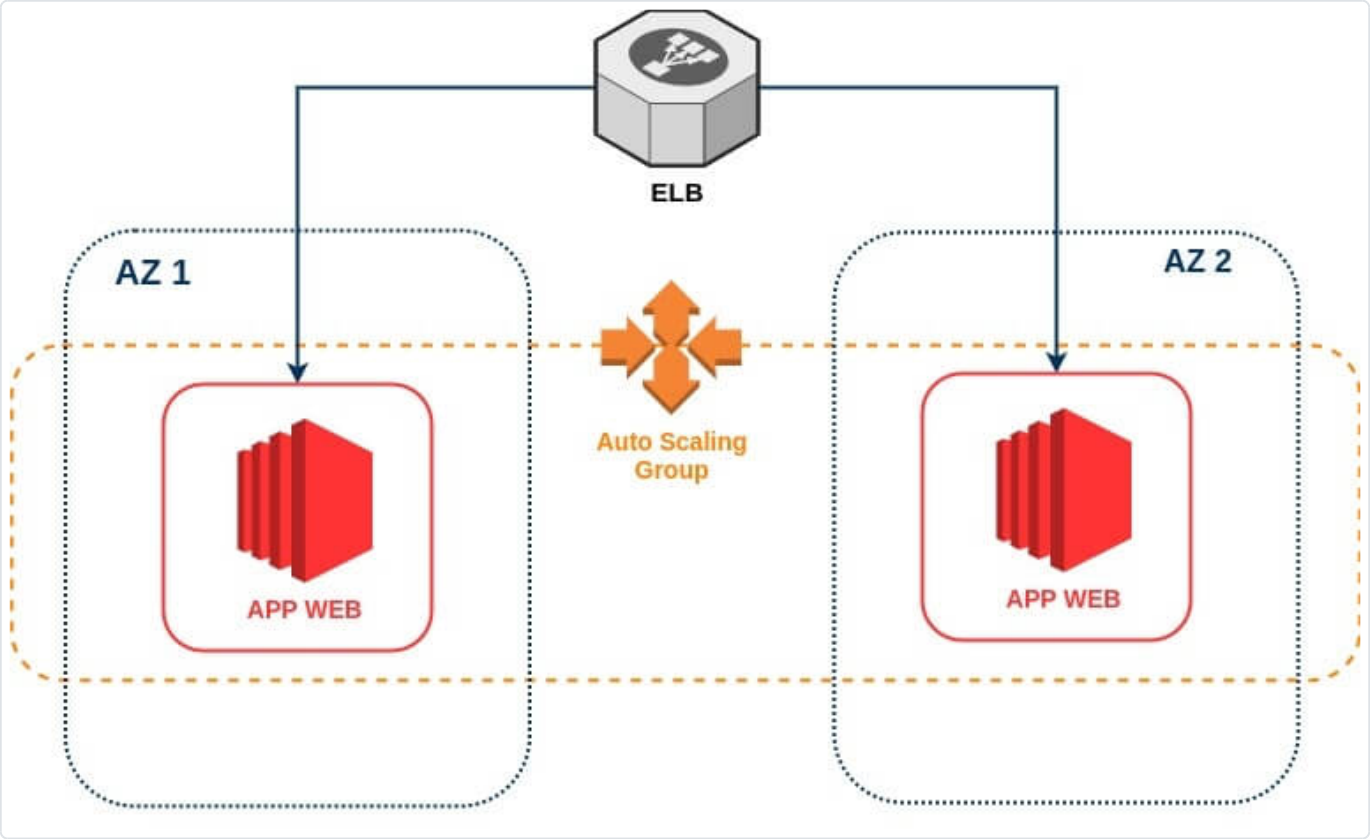
Sur AWS, pour créer un environnement hautement disponible, l'approche la plus recommandée consiste à utiliser plusieurs zones de disponibilité, comme ça dès qu'une zone de disponibilité tombe en panne notre application continuera de fonctionner sur notre seconde zone de disponibilité.

l'équilibreur de charge et l'AutoScaling Group

L'un des scénarios de construction d'une application Web qui doit être hautement disponible est d'utiliser le service d'équilibreur de charge d'AWS nommé ELB (Elastic Load Balancer) devant nos serveurs d'applications Web qui s'étendent sur plusieurs zones de disponibilité.

Ces serveurs webs doivent également être capable de **s'adapter automatiquement à différentes charges de travails** c'est-à-dire qu'un processus de mise à l'échelle de vos instances (augmentation et réduction de vos instances) en fonction du nombre de demandes reçues doit pouvoir se déclencher automatiquement. Ceci est possible avec le service AutoScaling Group (ASG), qui d'ailleurs s'intègre parfaitement avec le service ELB car il vous permet d'attacher les instances de votre ASG à un ou plusieurs équilibreurs de charge qui distribuera le trafic entrant entre ces instances. Ceci a pour effet de **maintenir des performances stables et prévisibles de la manière la plus rentable possible**.

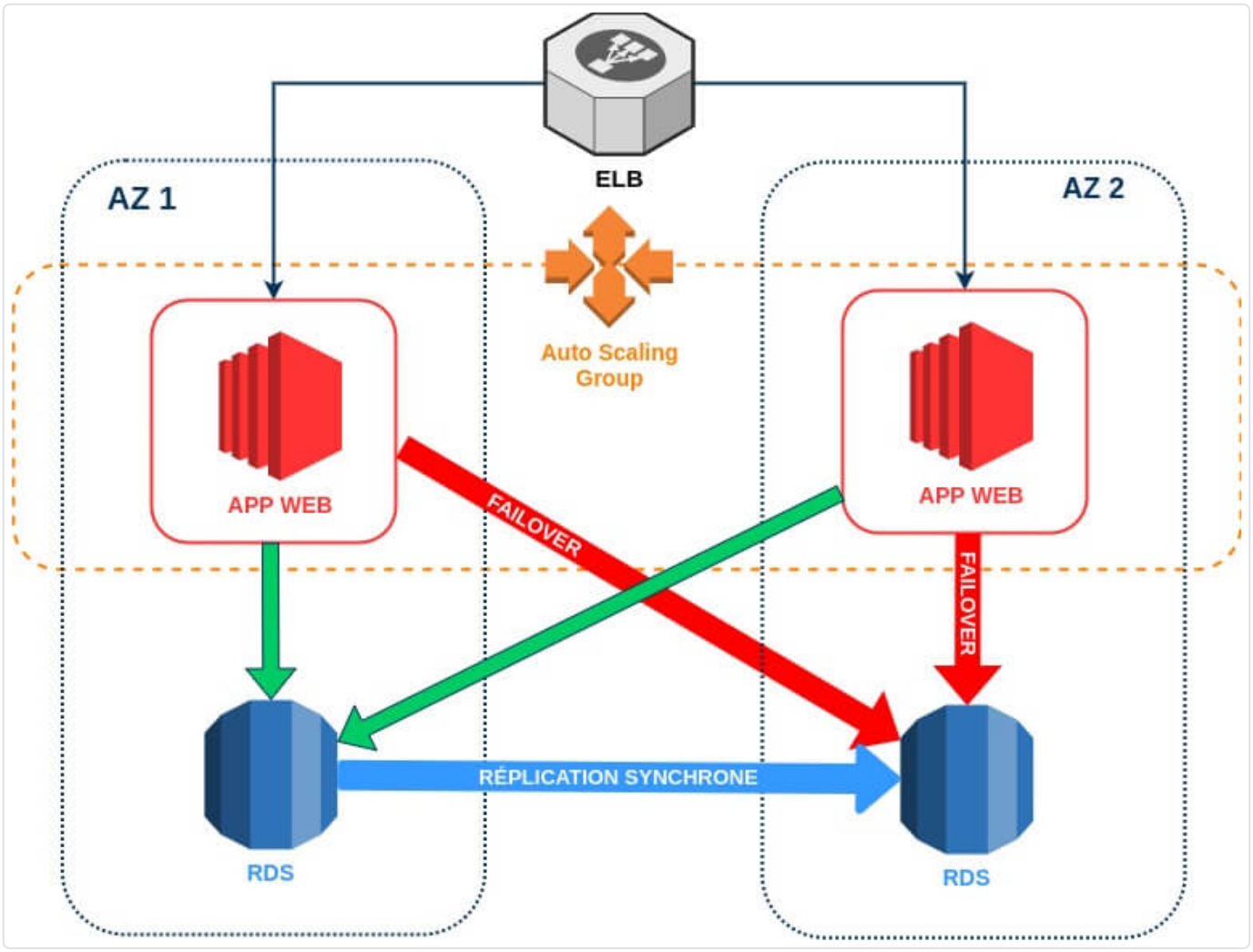
Jusqu'ici nous obtiendrons le schéma d'infrastructure suivant :



La base de données sur plusieurs AZs

L'approche simple pour avoir une haute disponibilité entre le niveau d'application Web et la base de données consiste à configurer un basculement automatique pour notre base de données qui est intrinsèquement pris en charge par le service RDS avec l'option "Multi-AZ". De cette façon votre base de données aura une **réplique de secours synchrone dans une zone de disponibilité différente**.

Pour le moment, voici un schéma qui explique la communication de nos instances EC2 avec le service RDS avec l'option "Multi-AZ" d'activée :



La partie réseau VPC

Pour contrôler la communication de nos ressources AWS, nous créerons un VPC (Virtual Private Cloud). un VPC vous permet de lancer des ressources AWS dans un réseau virtuel AWS que vous aurez défini. Ce réseau virtuel est logiquement isolé des autres réseaux virtuels dans le Cloud AWS. Dans ce VPC, vous spécifiez une plage d'adresses IP, des sous-réseaux et configurer des tables de routage.

Subnets

Un sous-réseau ("subnet" en anglais) est une plage d'adresses IP dans votre VPC où vous pouvez lancer dedans des ressources AWS. Vous avez deux types de subnets :

- **Subnet privé** : les services dans un sous-réseau privé ne sont pas accessibles depuis le réseau Internet.
- **Subnet public** : les services dans un sous-réseau public seraient accessibles à partir du réseau Internet, ce qui signifie que par défaut n'importe qui pourra atteindre les ressources dans ce subnet.

Dans notre scénario courant nous utiliserons une base de données, qui partagera des données avec nos serveurs Web. Nous utiliserons également un ELB qui redirigera le trafic vers ces mêmes serveurs WEB qui s'exécuteront dans le même VPC. Étant donné que notre instance de base de données doit uniquement communiquer avec nos serveurs Web et non sur Internet, elle doit être associée à un subnet privé. Dans ce même subnet privé, nous trouverons aussi nos instances web EC2, puisqu'elles ont besoin de communiquer qu'avec votre base de données et votre ELB et non avec l'extérieur (Internet).

Seul notre ELB sera hébergé dans un sous-réseau public, afin qu'il soit accessible depuis le réseau Internet. Il se chargera ensuite de rediriger les requêtes reçues vers l'un de nos subnets privés où se retrouvent nos instances EC2. privés. Vous l'aurez sans doute compris, cette limitation de communication offre une plus grande **sécurité de votre infrastructure**.

Internet GateWay

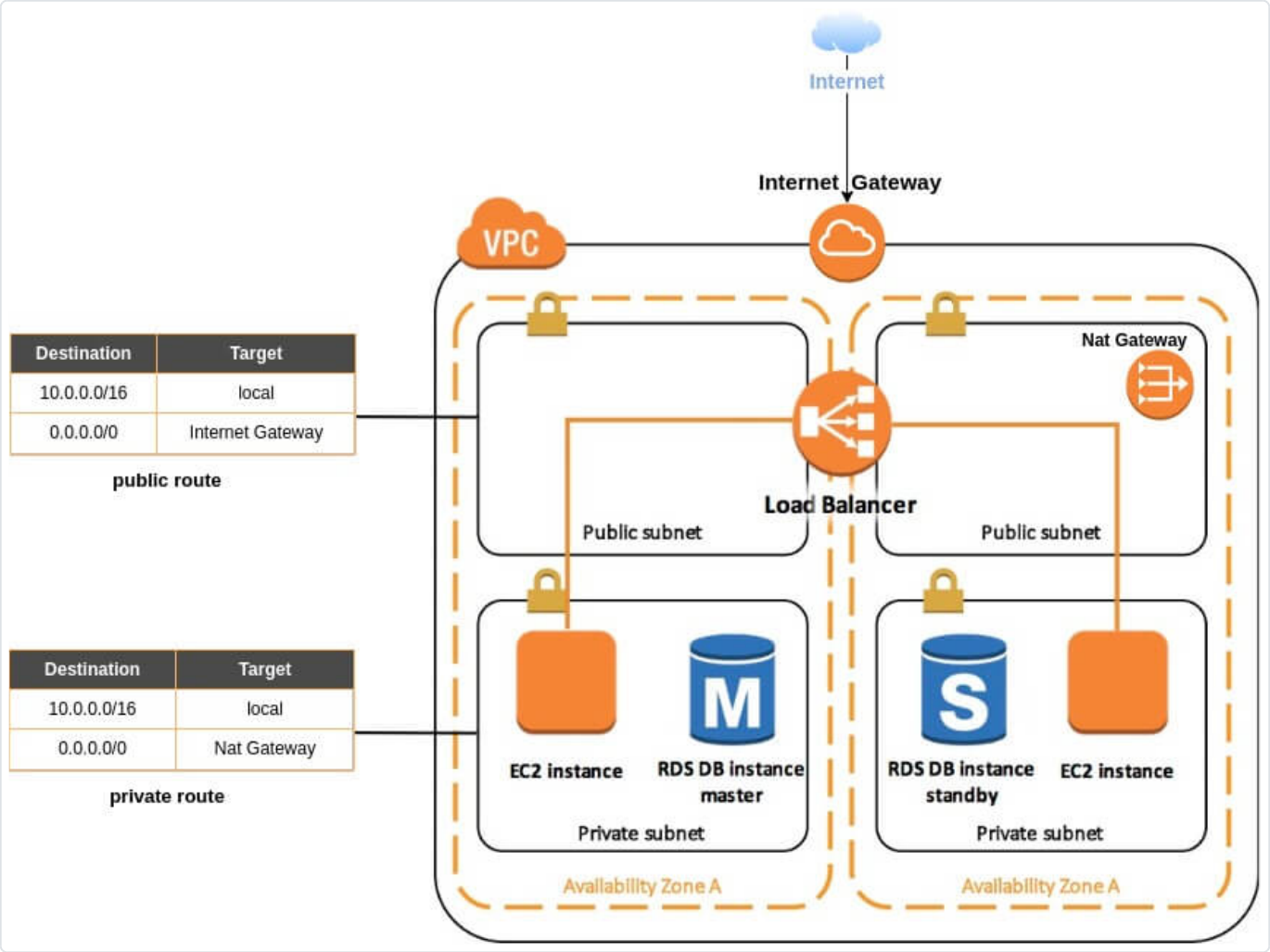
Pour qu'un subnet soit public, vous devez créer une passerelle Internet (Internet GateWay) et l'attachez à votre VPC. Vous devez ensuite créer une table de routage public avec une route de destination 0.0.0.0/0 vers cette passerelle Internet.

Nat GateWay

Cependant, nos serveurs Webs doivent être capables de télécharger certains packages (apache, php, etc ...) depuis le réseau Internet, mais comment faire cela tout en les gardant dans un subnet privé ? Nous pouvons utiliser dans ce cas une passerelle NAT ("Nat GateWay" en anglais) pour permettre aux instances de notre subnet privé de se connecter à Internet, mais empêcher tout de même le trafic provenant d'internet d'établir une connexion directe sur ces instances. Enfin, nous créerons une table de routage privée avec une route de destination 0.0.0.0/0 vers votre passerelle NAT.

Schéma

En réunissant tous ces composants nous obtiendrons le schéma d'infrastructure suivant :



Les autres services AWS

S3 et rôle IAM

Pour ce tp, je vous propose de télécharger les sources de notre application web que j'ai codée en php en cliquant ici. Je vous demanderai de déposer depuis Terraform les sources automatiquement dans un bucket S3 que vous aurez préalablement créé et de les récupérer ensuite automatiquement dans vos instances EC2.

Pour cela je vous suggère d'utiliser les rôles IAM qui sont utilisés pour déléguer automatiquement l'accès à quelques ressources AWS. À l'inverse d'un utilisateur

IAM qui aura quant à lui besoin d'un ID et d'une clé secrète que vous devez enregistrer sur chacune de vos instances web, ce qui n'est pas le cas du rôle. Car en effet, vous aurez besoin attacher uniquement votre rôle à votre instance EC2 et elle aura automatiquement accès à votre ressource AWS selon la policy que vous aurez spécifiée dans votre rôle, sans la nécessité de spécifier votre ID et votre clé secrète

Les alarmes CloudWatch

CloudWatch est le service de surveillance des ressources et applications AWS. Nous l'utiliserons avec notre ASG pour déclencher une alarme dès lors que nos instances auront atteint un seuil maximal ou minimal d'utilisation du CPU.

Les instances EC2 d'un ASG envoient périodiquement des métriques avec une granularité d'une à cinq minutes sur le service CloudWatch. En procédant ainsi, les métriques récupérées peuvent déclencher des alarmes pour effectuer des opérations automatiques sur votre ASG afin de pouvoir **répondre rapidement aux modifications de vos charges de travail**. On peut par exemple ainsi scale up nos instances (augmenter le nombre de nos instances) de notre ASG à partir d'une utilisation de 80% du CPU et scale down (diminuer le nombre d'instances) à partir de 20% d'utilisation du CPU.

Suggestions de procédure à suivre

Il suffit maintenant de rassembler tous ces éléments pour créer votre infrastructure. Je vous propose ci-dessous quelques suggestions de procédure à suivre pour créer votre infrastructure. Libre à vous de la respecter entièrement ou partiellement.

VPC

Voici la procédure suggérée pour la partie réseau :

- Création d'un VPC avec un bloc CIDR IPv4 de 10.0.0.0/16.
- Création de deux subnets publics sur deux zones de disponibilité différentes avec deux blocs CIDR IPv4 de 10.0.1.0/24 et 10.0.2.0/24.
- Création de deux subnets privés sur les mêmes zones de disponibilités que les subnets publics avec deux blocs CIDR IPv4 de 10.0.3.0/24 et 10.0.4.0/24.
- Création de la passerelle Internet.
- Création d'une adresse IP statique avec le service IP Elastic à attacher sur votre passerelle NAT.
- Création de la passerelle NAT.
- Création d'une table de routage publique et privée à associer à vos subnets respectifs.
- Création d'une route de destination 0.0.0.0/0 vers votre passerelle Internet dans votre table de routage publique.
- Création d'une route de destination 0.0.0.0/0 vers votre passerelle NAT dans votre table de routage privée.

S3 et rôle IAM

Voici la procédure suggérée pour créer votre rôle IAM et votre bucket S3 :

- Création d'un bucket S3 avec un nom unique et un ACL prêt à l'emploi de type "private", ainsi personne d'autre ne possédera les droits d'accès sur votre bucket (sauf le propriétaire). Vous devez également trouver un moyen de

rajouter les sources de l'application web automatiquement sur votre bucket .

- Création d'un rôle attaché aux services EC2 avec une policy qui autorise un accès complet aux services S3.
- Création d'un profil d'instance pour transmettre les informations liées au rôle créé précédemment à vos instances EC2 lorsque celles-ci démarrent.

Vous pouvez utiliser les VPCs Endpoints pour se connecter à votre service S3 à l'aide d'un réseau privé au lieu du réseau internet. Vous avez également la possibilité de créer un hôte bastion qui n'est rien d'autre qu'une instance EC2 sur votre subnet public qui est autorisée à se connecter via le protocole SSH à vos instances EC2 situées dans vos subnets privés.

ELB et ASG

Voici la procédure suggérée pour créer votre équilibreur de charge et votre AutoScaling Groups :

- Création d'un groupe de sécurité ("Security Group" en anglais) pour votre ASG autorisant uniquement le trafic provenant de votre ELB sur le port 80.
- Création d'un groupe de sécurité pour votre ELB autorisant uniquement le trafic provenant d'internet sur le port 80.
- Création d'une Target Group sur le port 80 afin d'aider votre ELB à acheminer les requêtes HTTP vers les instances de votre ASG.
- Création de votre équilibreur de charge de type "Application" attaché à votre subnet public et votre groupe de sécurité ELB.

- Création d'un HTTP Listener attaché à votre ELB et Target Group afin de déterminer la façon dont votre équilibreur de charge acheminera les demandes vers les cibles enregistrées dans votre Target Group.
- Création d'une configuration de lancement ("launch configuration" en anglais) de votre ASG où vous spécifierez l'ami, type d'instance ("t2.micro" pour rester dans l'offre gratuite d'AWS), profil d'instance avec votre rôle IAM, user-data, paire de clés, groupe de sécurité à utiliser sur les instances de votre ASG.
- Création de votre AutoScaling Groups où vous spécifierez la configuration de lancement créé précédemment, les subnets privés sur lesquels se lanceront vos instances, votre Target Group, un contrôle de type "ELB" et le nombre désiré/minimum/maximum de vos instances.

Pour que votre instance ec2 soit déjà prête et configurée instantanément lors d'un scale up de vos instances EC2 de votre ASG, vous avez le choix entre créer et utiliser une ami personnalisée ou tout créer et configurer depuis le user-data. Pour ce tp, j'ai utilisé la méthode du user-data afin de vous montrer concrètement les commandes exécutées lors de la création de mes nouvelles instances EC2.

RDS

Voici la procédure suggérée pour créer votre base de données relationnelle :

- Création d'un groupe de sécurité autorisant uniquement le trafic provenant de vos instances web sur le port 3306.
- Création de votre base de données de type "mariadb" en utilisant le service RDS. Elle sera attachée à votre subnet privé et le groupe de sécurité créé antérieurement. Assurez-vous aussi, que votre instance MariaDB possède une

classe de type "db.t2.micro", 20 Go de stockage maximum et des sauvegardes automatisées activées avec une période de rétention d'un jour afin que vous restiez éligible à l'offre gratuite d'AWS. Enfin, vérifiez que l'option "Multi-AZ" est activée.

Alarme CloudWatch

Voici la procédure suggérée pour créer votre alarme CloudWatch basée sur l'utilisation moyenne du processeur :

- Création de deux stratégies d'AutoScaling, une pour le scale up et une autre pour le scale down. Les stratégies seront de type "simple" afin d'augmenter ou diminuer la capacité actuelle de notre ASG en fonction d'un seul ajustement (ex : utilisation du CPU). L'ajustement sera de type "ChangeInCapacity" qui aura comme valeur "1" pour la stratégies du scale up dans le but de rajouter une seule instance et "-1" pour celle du scale down pour supprimer qu'une seule instance.
- Création de deux alarmes CloudWatch, l'une sera basera sur la métrique "CPUUtilization" avec un seuil d'utilisation supérieur ou égale à 80% d'utilisation du CPU pour déclencher le stratégie ASG scale up et une autre pour une utilisation inférieure à 5% pour déclencher la stratégie ASG scale down.

Vous pouvez aller plus loin en créant des notifications par mail depuis le service SNS d'AWS lorsqu'une des stratégies d'ASG se déclenche.

Les sources de notre application et exigences du TP

Cette application codée en php permet tout simplement de poster un article qui est ensuite sauvegardé sur notre base de données mysql. Voici quelques indications sur deux fichiers sources qui vous devez obligatoirement prendre en considération :

- **db-config.php** : contient la configuration requise pour que votre application communique avec votre base de données, vous retrouverez :
  - `##DB_HOST##` : à remplacer par l'ip ou le nom dns de votre base de données.
  - `##DB_USER##` : à remplacer par le nom d'utilisateur de votre base de données.
  - `##DB_PASSWORD##` : à remplacer par le mot de passe utilisateur de votre base de données.
- **articles.sql** : contient la requête SQL à exécuter pour créer l'architecture de votre table dans votre base de données.

Informations importantes avant de commencer le TP, je vous demanderai de **créer vos ressources Terraform sous forme de modules** ! Good Luck .

Solution

J'espère que vous avez réussi à réaliser ce tp. Même si vous n'avez pas forcément réussi à tout faire. L'essentiel c'est qu'au moins quelques ressources soient déployées, car ça restera toujours mieux que de voir la solution directement sans même essayer.

Vous pouvez commencer par télécharger les sources de ma solution depuis mon repository github.

Ci-dessous, je vous présente quelques détails de ma solution. Le but n'est pas d'expliquer toutes les lignes de code mais de vous éclairer sur les choix et astuces que j'ai appliquées. D'ailleurs n'hésitez pas à modifier le code selon votre guise et de le partager dans l'espace commentaires avec un lien de votre repository github ou autres !

Arborescence choisie

Voici comment j'ai organisé l'arborescence de mon projet :

```
|___ modules/
|   |___ alb_asg/
|   |___ cloudwatch_cpu_alarms/
|   |___ ec2_role_allow_s3/
|   |___ rds/
|   |___ s3/
|
|___ vpc/
|___ src/
|___ keys/
|___ vars.tf
|___ main.tf
|___ outputs.tf
|___ README.md

|___ .gitignore
```

- **modules** : répertoire pour y héberger nos différents modules.
- **src** : répertoire pour y héberger les sources de notre application qui seront par la suite envoyées sur notre bucket S3.
- **keys** : répertoire pour y héberger la paire de clé SSH, au cas où nous aurons besoin de nous connecter sur nos instances EC2.
- **vars.tf** : fichier de variables du module racine.
- **main.tf** : fichier de configuration principale du module racine.

- **outputs.tf** : fichier de variables de sortie du module racine.
- **README.md** : fichier de documentation principale de notre projet.
- **.gitignore** : fichier contenant une liste de fichiers/dossiers à ignorer lors d'un commit.

Module vpc

Voici les variables utilisées pour mon module vpc :

```
variable "vpc_cidr" {
  default = "10.0.0.0/16"
}

variable "public_subnets_cidr" {
  type    = list
  default = ["10.0.1.0/24", "10.0.2.0/24"]
}

variable "private_subnets_cidr" {
  type    = list
  default = ["10.0.3.0/24", "10.0.4.0/24"]
}

variable "azs" {
  type        = list
  description = "AZs to use in your public and private subnet  (make sure they are co
  default     = ["us-west-2a", "us-west-2b"]
}

variable "prefix_name" {}
```

J'autorise ici l'utilisateur final le choix de choisir son propre bloc CIDR IPv4 pour son VPC, ses subnets privés et publics. Il peut également spécifier les zones de disponibilités à utiliser sur ces mêmes subnets. Vous verrez également tout au long du code de ce projet, la référence à la variable `prefix_name` qui comme son nom l'indique sera le préfixe à utiliser lors du nommage de nos différentes ressources AWS.

Voici ensuite une partie de la configuration de notre module qui démontre comment j'ai **évité la duplication de code** grâce à l'utilisation du système de boucles :

```
...
...

# Public subnets
resource "aws_subnet" "main-public" {
  count                   = length(var.public_subnets_cidr)
  vpc_id                  = aws_vpc.main.id
  cidr_block              = var.public_subnets_cidr[count.index]
  map_public_ip_on_launch = true
  availability_zone       = var.azs[count.index]

  tags = {
    Name = "${var.prefix_name}-public-${count.index}"
  }
}

# Internet GW
resource "aws_internet_gateway" "main-gw" {
  vpc_id = aws_vpc.main.id

  tags = {
    Name = "${var.prefix_name}"
  }
}

# Public route tables
resource "aws_route_table" "main-public" {
  vpc_id = aws_vpc.main.id

  route {
    cidr_block = "0.0.0.0/0"
    gateway_id = aws_internet_gateway.main-gw.id
  }

  tags = {
    Name = "${var.prefix_name}-public"
  }
}

# Route associations public
resource "aws_route_table_association" "main-public" {
  count          = length(var.public_subnets_cidr)
  subnet_id      = aws_subnet.main-public[count.index].id
  route_table_id = aws_route_table.main-public.id
}

...
...
```

J'utilise le paramètre `count` pour boucler sur ma liste de blocs CIDR IPv4 de mes subnets publics en récupérant la longueur cette liste depuis la fonction `length(ma_liste)`, ensuite pour chaque itération je récupère l'élément de la liste `public_subnets_cidr` et `azs` depuis l'attribut `index` de mon paramètre `count`. J'utilise le même processus pour associer ma table de routage publique à mes subnets publics. Cette manipulation évite comme précisé plus haut, la duplication de code.

Passons maintenant aux variables de sortie de notre module vpc :

```
output "vpc_id" {
  value = aws_vpc.main.id
}

output "private_subnet_ids" {
  value = [for private_subnet in aws_subnet.main-private : private_subnet.id]
}

output "public_subnet_ids" {
  value = [for public_subnet in aws_subnet.main-public : public_subnet.id]
}
```

Je rajoute les attributs de mes ressources que je vais récupérer et utiliser sur mes autres modules, notamment l'id de mon vpc et ceux de mes subnets publics et privés. Autre astuce, J'utilise la boucle `for` pour créer une variable de type `list` des ids de mes subnets car c'est le type qui sera demandé plus tard dans mes autres modules de ressources.

Module s3

Voici les variables demandées du module S3 :

```
variable "bucket_name" {
  description = "bucket name must be globally unique"
}
```

```
variable "path_folder_content" {
  description = "add the contents of a folder to your S3 bucket (relatif or absolute
}
```

D'un côté je demanderai à l'utilisateur de spécifier un nom de bucket qui doit être unique sur l'ensemble des noms de bucket mondiaux d'Amazon S3 et le chemin local des sources de l'application web qu'il souhaite envoyer dans son bucket S3.

Au jour d'aujourd'hui, il est compliqué depuis Terraform d'envoyer le contenu d'un dossier entier dans un bucket S3. J'ai dû donc trouver une astuce pour contourner ce problème. Pour ce faire, j'ai utilisé le provisionneur `local-exec` avec la commande `aws s3 sync` qui permet de copier et mettre à jour récursivement les nouveaux fichiers du répertoire source vers la destination, soit :

```
# S3 Bucket
resource "aws_s3_bucket" "my_bucket" {
  bucket = var.bucket_name
  acl    = "private"
  tags = {
    Name = var.bucket_name
  }
}

# Add content to bucket
resource "null_resource" "add_src_to_s3" {
  triggers = {
    build_number = "${timestamp()}" # run it all times
  }
  provisioner "local-exec" {
    command = "aws s3 sync ${var.path_folder_content} s3://${var.bucket_name}/"
  }
  depends_on = [aws_s3_bucket.my_bucket]
}
```

Vous remarquerez, que j'ai rajouté l'argument `triggers` afin de forcer réexécution de l'ensemble du provisionneur en renvoyant la date et l'heure actuelles depuis la fonction `timestamp()` qui seront donc différents pour chaque nouvelle exécution de notre module de façon à s'assurer que notre bucket s3 contient toujours la dernière version de nos sources.

Module ec2_role_allow_s3

Pour le bon fonctionnement de ce module j'ai besoin de récupérer le nom du bucket afin de créer la stratégie de mon rôle :

```
variable "bucket_name" {}

variable "prefix_name" {}
```

Ensuite dans ma configuration Terraform j'ai suivi la procédure exposée plus haut afin de créer mon profile d'instance que j'exporte dans mon fichier de variables de sortie **outputs.tf** :

```
output "name" {
  value = aws_iam_instance_profile.s3-mybucket-role-instanceprofile.name
}
```

Module alb_asg

Nous passons maintenant à la partie la plus complexe de notre projet, soit la création de notre ELB et ASG.

On commence par récupérer l'id de notre VPC et de nos subnets ainsi que des informations basiques pour la création de notre ASG (ami, type d'instance, nombre min/max/désiré d'instances, profile d'instance, clé publique), le port et le protocole à utiliser dans notre groupe de sécurité et dans le Listener de notre ALB.

```
variable "prefix_name" {}

variable "vpc_id" {}

variable "private_subnet_ids" {
  type = list
}

variable "public_subnet_ids" {
  type = list
```

```
}

variable "webserver_port" {
 default = 80
}

variable "webserver_protocol" {
 default = "HTTP"
}

variable "user_data" {}

variable "instance_type" {
  default = "t2.micro"
}

variable "role_profile_name" {}


variable "min_instance" {
  description = "minimum number of instances for your ASG"
  default     = 2
}

variable "desired_instance" {
  description = "starting number of instances for your ASG"
  default     = 2
}

variable "max_instance" {
  description = "maximum number of instances for your ASG"
  default     = 4
}

variable "ami" {}

variable "path_to_public_key" {
  description = "relatif or absolute path of your public ssh key"
}
```

Pour la partie configuration de ce module, il était demandé d'autoriser uniquement le trafic provenant de notre ELB pour le groupe de sécurité de nos instances web. L'astuce pour satisfaire cette exigence, consiste non pas à utiliser l'adresse IP de notre ELB (c'est possible mais non recommandé) mais bel et bien le groupe de sécurité de notre ELB en tant que cible dans la règle ingress de notre groupe de sécurité de nos instances web. Ceci est possible depuis l'argument `security_groups` :

```
# Security group for ALB
resource "aws_security_group" "sg-alb" {
  vpc_id        = var.vpc_id
  ...
  ...

  ingress {
    from_port   = var.webserver_port
    to_port     = var.webserver_port
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
  ...
  ...

# Security group for ASG instances
resource "aws_security_group" "sg-instances" {
  vpc_id            = var.vpc_id
  ...
  ...

  ingress {
    from_port       = var.webserver_port
    to_port         = var.webserver_port
    protocol        = "tcp"
    security_groups = [aws_security_group.sg-alb.id]
  }

  ...
  ...
}
```

Dans ma configuration de lancement de mon ASG j'utilise l'option `create_before_destroy`, cet indicateur est utilisé pour garantir que le remplacement d'une ressource est créé avant la destruction de l'instance d'origine :

```
# ASG launch configuration
resource "aws_launch_configuration" "my-launchconfig" {
  ...
  ...

  lifecycle {
    create_before_destroy = true
  }
}
```

Lors de la création de l'ASG, j'ai ajouté l'option `force_delete` qui permet de modifier le comportement par défaut de destruction de Terraform. Normalement, Terraform attend que tout votre pool instances EC2 soit terminé avant de supprimer votre ASG. Cependant, si vous valorisez cet attribut à `true`, cela forcera la suppression de votre ASG sans attendre la mise en fin de toutes les instances de votre ASG.

Ensuite, j'ai suivi la procédure énoncée plus haut pour notre ELB. Enfin, dans mes variables de sortie, j'exporte le nom DNS de mon ELB afin de pouvoir tester son bon fonctionnement directement depuis mon navigateur. J'exporte également le nom mon ASG qui sera utilisé plus tard dans nos alarmes CloudWatch, ainsi que l'id du groupe de sécurité de nos instances web afin de le rajouter en tant que cible dans la règle ingress du groupe de sécurité de notre base de données :

```
output "alb_dns_name" {
  value = aws_lb.my-alb.dns_name
}

output "webserver_sg_id" {
  value = [aws_security_group.sg-instances.id]
}

output "asg_name" {
  value = aws_autoscaling_group.my-autoscaling.name
}
```

Module rds

Dans le fichier variables de ce module, il sera réclamé à l'utilisateur final des renseignements sur la partie réseau comme l'id du vpc, du subnet privé et celui du groupe de sécurité de nos instances Web. Il doit également définir le nom, nom d'utilisateur, mot de passe et la version de la base de données qui sera de type mariadb, ainsi que des informations sur les capacités de l'instance à utiliser, c'est-à-

dire le type d'instance/stockage, période de rétention, l'espace de stockage, l'activation ou non du "Multi-AZ". Ce qui nous donne le contenu suivant :

```
variable "prefix_name" {}

variable "vpc_id" {}

variable "private_subnet_ids" {
 type = list
}

variable "webserver_sg_id" {
  type        = list
  description = "security groupd id of the webserver"
}

variable "storage_gb" {
 description = "how much storage space do you want to allocate?"
   default     = 5
}

variable "mariadb_version" {
 default = "10.1.34"
}

variable "mariadb_instance_type" {
 description = "use micro if you want to use the free tier"
 default     = "db.t2.small"
}

variable "db_name" {
 description = "database name"
}

variable "db_username" {
 description = "database username"
 default     = "root"
}

variable "db_password" {
 description = "database user password"
}

variable "is_multi_az" {
 description = "set to true to have high availability"
 default = false
}

variable "storage_type" {
 description = "Storage type used for the database"
 default = "gp2"
```

```
}


variable "backup_retention_period" {
 description = "how long you’re going to keep your backups (30 max) ?"
 default     = 1
}
```

Dans la configuration de votre module, si vous jamais vous souhaitez configurer votre base de données mariadb en définissant par exemple une longueur maximale de paquet à envoyer ou à recevoir avec le paramètre `max_allowed_packet` de mariadb, c'est possible mais sachez juste que vous n'aurez aucun accès SSH directe sur l'instance hébergeant votre base de données. Pour ce faire, vous devez créer un groupe de paramètres qui sera automatiquement pris en compte lors de la création de l'instance de votre base de données. Voici l'exemple que j'utilise dans mon code :

```
# Parameters of the db (mariadb)
resource "aws_db_parameter_group" "mariadb-parameters" {
  name        = "mariadb-params"
  family      = "mariadb10.1"
  description = "MariaDB parameter group"

  # Parameters example
  parameter {
    name  = "max_allowed_packet"
    value = 16777216
  }
}

# Db instance
resource "aws_db_instance" "mariadb" {
  ...
  ...
  parameter_group_name      = aws_db_parameter_group.mariadb-parameters.name
  ...
  ...
}
```

Les variables de sortie qui me seront nécessaires plus tard, sont l'adresse DNS et le nom d'utilisateur de ma base de données :

```
output "host" {
  value = aws_db_instance.mariadb.address
}

output "username" {
  value = aws_db_instance.mariadb.username
}
```

Module cloudwatch_cpu_alarms

Pour notre dernier module, je propose à l'utilisateur final de choisir le seuil à atteindre sous forme de pourcentage pour déclencher notre alarme CloudWatch de scale up et scale down. je lui demande également de renseigner le nom de l'ASG sur lequel sera déclencher le processus du scale up/down :

```
variable "prefix_name" {}

variable "max_cpu_percent_alarm" {
 description = "percentage of CPU consumption to achieve scale up"
 default  = 80
}

variable "min_cpu_percent_alarm" {
 description = "percentage of CPU consumption to achieve scale down"
 default  = 5
}

variable "asg_name" {
 description = "ASG name used to increase or decrease ec2 instances"
}
```

Dans la configuration de ce module, j'ai suivi à la lettre la procédure décrite plus haut pour nos alarmes CloudWatch. Cependant, je tiens à préciser que si jamais vous souhaitez être notifiés par mail avec le service SNS lors d'un dimensionnement dynamique de votre ASG. Sachez que Terraform vous donnera la possibilité de créer un topic dans le service SNS pour ce type de demande, mais malheureusement sur la version actuelle de cet article (v0.12.24), Terraform ne vous permet pas de renseigner une adresse mail d'abonnement à ce topic. C'est donc une opération

qu'il faut effectuer depuis la console AWS en post création de votre infrastructure.

Néanmoins, je vous propose ci-dessous le code à utiliser pour créer votre topic SNS :

```
# Autoscaling notifications
resource "aws_sns_topic" "my-sns" {
 name         = "${var.prefix_name}-asg-sns"
 display_name = "my ASG SNS topic"
}

resource "aws_autoscaling_notification" "my-notify" {
 group_names = ["${var.asg_name}"]
 topic_arn     = "${aws_sns_topic.my-sns.arn}"
 notifications  = [
   "autoscaling:EC2_INSTANCE_LAUNCH",
   "autoscaling:EC2_INSTANCE_TERMINATE",
   "autoscaling:EC2_INSTANCE_LAUNCH_ERROR"
 ]
}
```

Module racine

Pour le module racine, voici à quoi va ressembler mon ficher de variables :

```
variable "region" {
  default = "us-west-2"
}

variable "prefix_name" {
  default = "devopssec"
}

variable "bucket_name" {
  default = "devopssec-bucket"
}

variable "db_password" {}

data "aws_ami" "ubuntu-ami" {
    most_recent = true

    filter {
        name   = "name"
        values = ["ubuntu/images/hvm-ssd/ubuntu-bionic-18.04-amd64-server-20200430"]
    }
    owners = ["099720109477"] # Canonical
}
```

Les variables `prefix_name` et `bucket_name` seront utilisées plusieurs fois en tant que module d'où leurs présences dans le fichier **variables.tf**. La variable `db_password` est une donnée secrète que je vais définir hors mon module racine d'où l'absence de la valeur par défaut. Enfin, grâce à ma Data Source `ubuntu-ami` je récupère automatiquement l'ami de la version 18.04 LTS bionic (amd64) de la région us-west-2 (plus d'informations sur le chapitre consacré aux Data Sources).

Dans le fichier de configuration Terraform de mon module racine, voici le user-data que j'utilise pour installer et configurer mon application web :

```
module "my_alb_asg" {
    source               = "./modules/alb_asg"
    ...
    ...
    user_data =
```

Dans ce user-data, les tâches suivantes seront exécutées :

- Mise à jour de la liste des packages de notre distribution.
- Installation du service web apache, l'interpréteur php avec la bibliothèque mysql et la cli (Command Line Interface) d'AWS.
- Activation permanente du service apache via la commande systemctl.
- Suppression de la page web d'accueil par défaut **index.html**.
- Téléchargement de nos sources depuis notre bucket S3.
- Création de l'architecture de la table de notre base de données mariadb.
- Configuration de la partie base de données de notre application web depuis la commande sed.

Pour finir, dans le fichier de variables de sortie de mon module racine, je récupère le nom DSN public de notre ELB :

```
output "alb_dns_name" {
  value = module.my_alb_asg.alb_dns_name
}
```

Lancement et test du projet

Pour utiliser ce projet, il faut d'abord commencer par indiquer le mot de passe root de notre base de données. Dans mon cas, j'ai choisi de le définir dans un fichier nommé **terraform.tfvars** (pour rappel ce nom de fichier est automatiquement pris en charge par Terraform lors de l'exécution de votre configuration) que je vais d'ailleurs ignorer dans mon fichier **.gitignore**. Voici à quoi ressemble ce fichier :

```
db_password = "votre-mot-de-passe"
```

Vous devez par la suite créer votre paire de clés ssh dans le dossier **keys**. Dans mon cas, j'utilise la commande ssh-keygen comme suit :

```
ssh-keygen -t rsa

Generating public/private rsa key pair.
Enter file in which to save the key (/home/hatim/.ssh/id_rsa): ./keys/terraform
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
```

Rajoutez ensuite le chemin de votre clé publique dans le paramètre **path_to_public_key** du module **my_alb_asg**. Dans mon cas ça sera :

```
module "my_alb_asg" {
    ...
    ...
    path_to_public_key   = "/home/hatim/Documents/tmp/terraform/sources/keys/terrafo
    ...
    ...
```

```
}
```

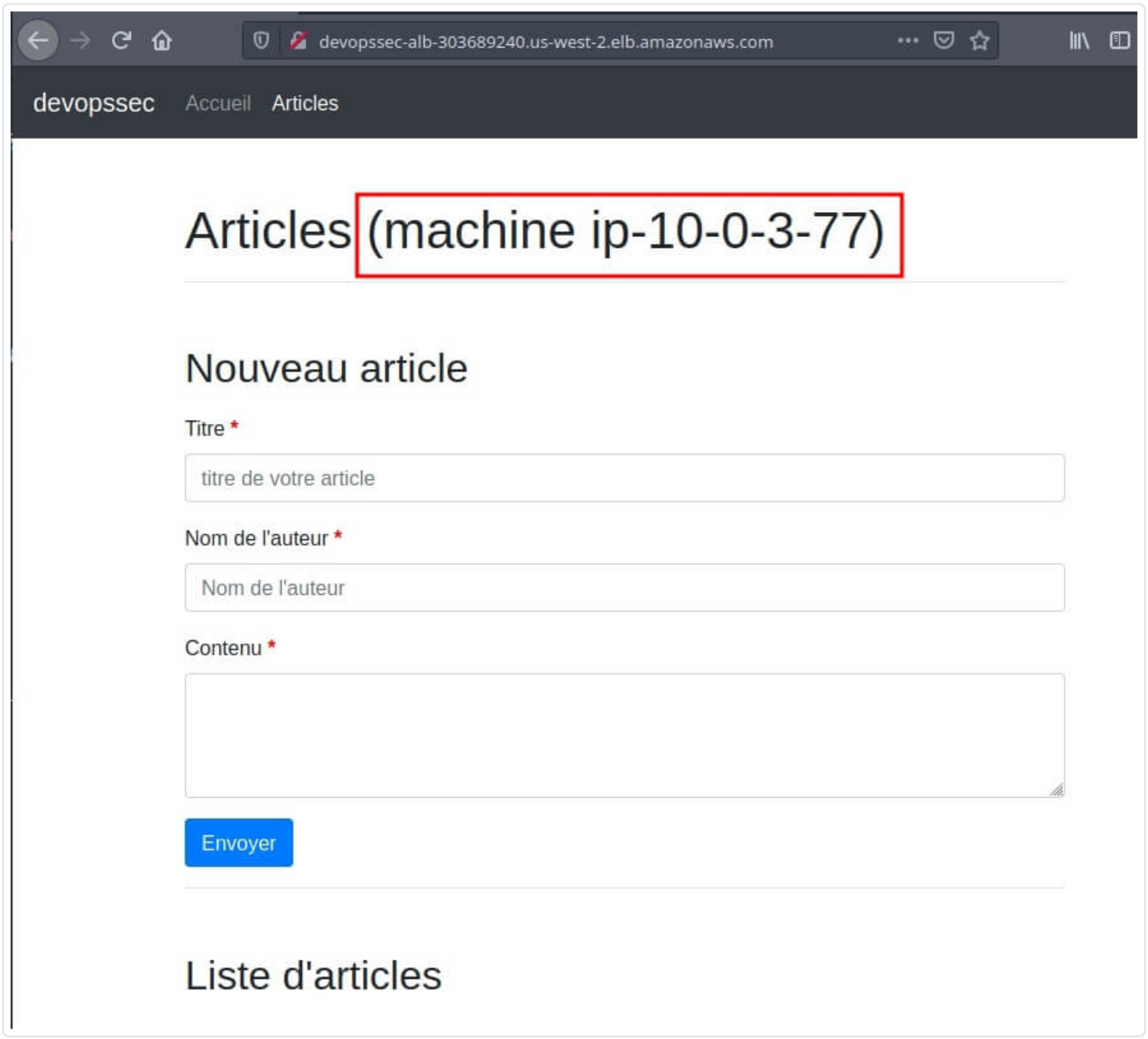
Lançons ensuite notre projet avec la commande suivante :

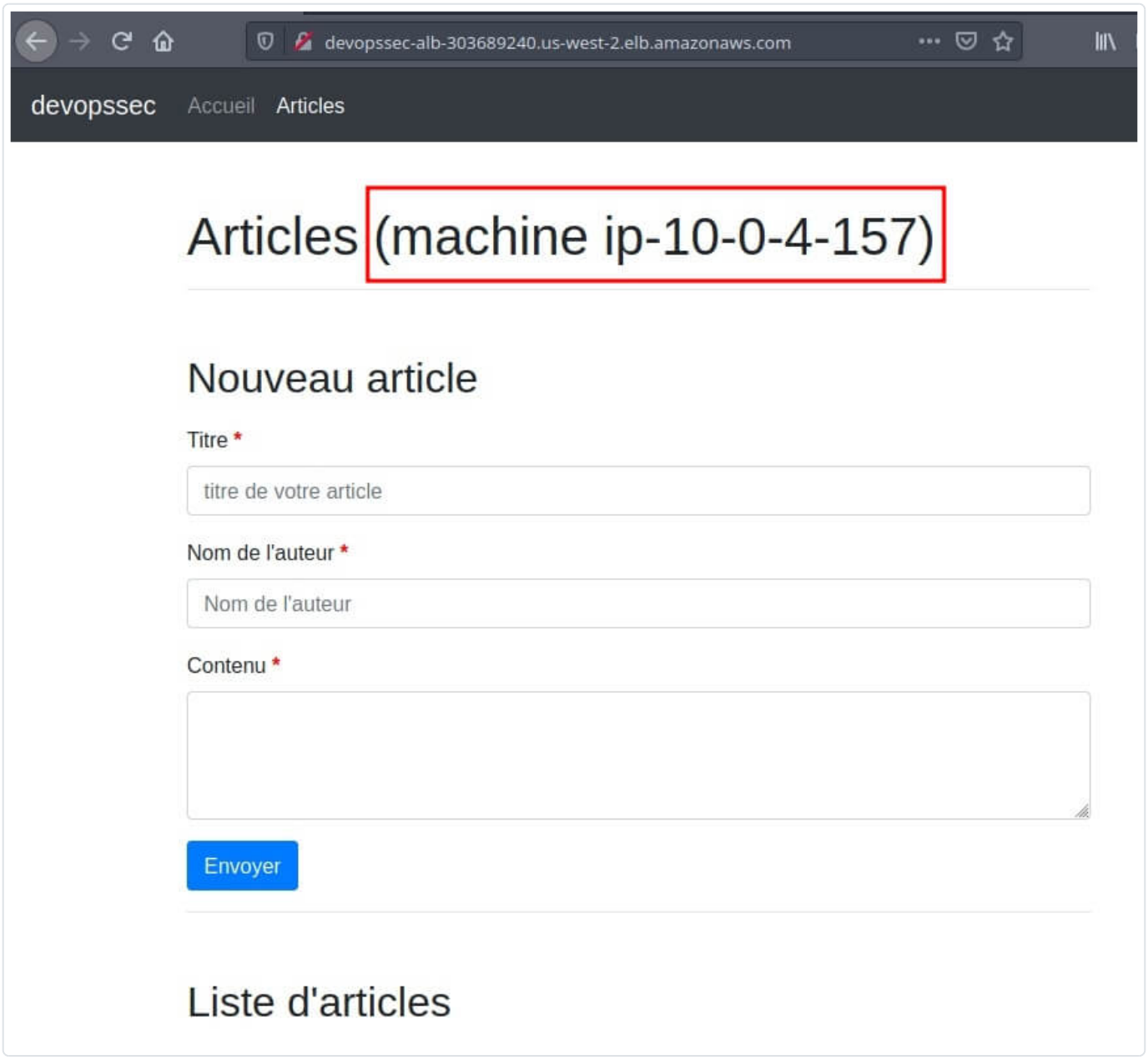
```
terraform init && terraform apply
```

Résultat :

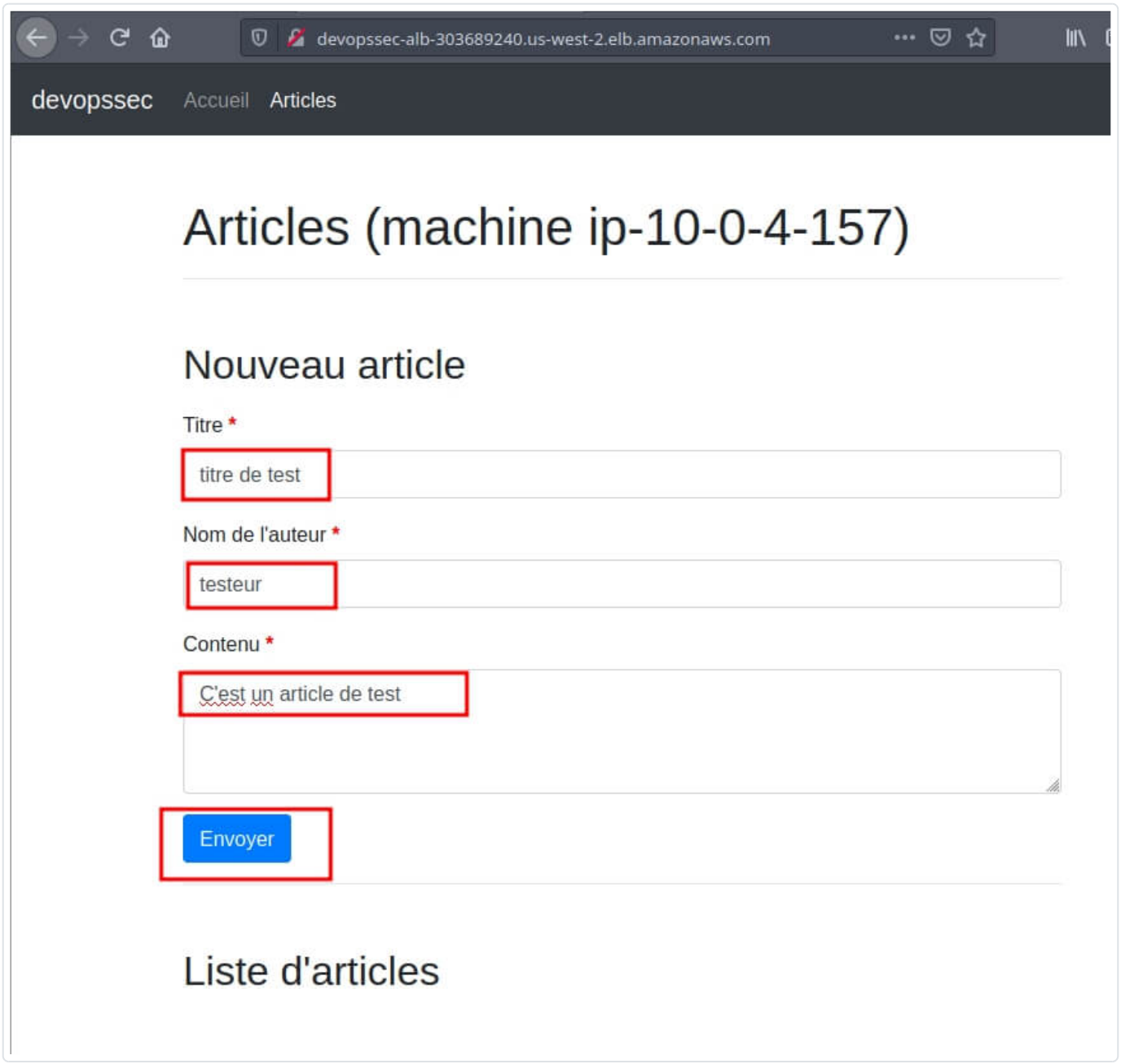
```
...
...

Outputs:

alb_dns_name = devopssec-alb-303689240.us-west-2.elb.amazonaws.com
```

Pour vérifier que notre ELB redirige le trafic sur les différentes instances web de notre ASG, j'ai rajouté le nom de la machine dans le code php. Ce qui nous donne la page web suivante sur notre navigateur :

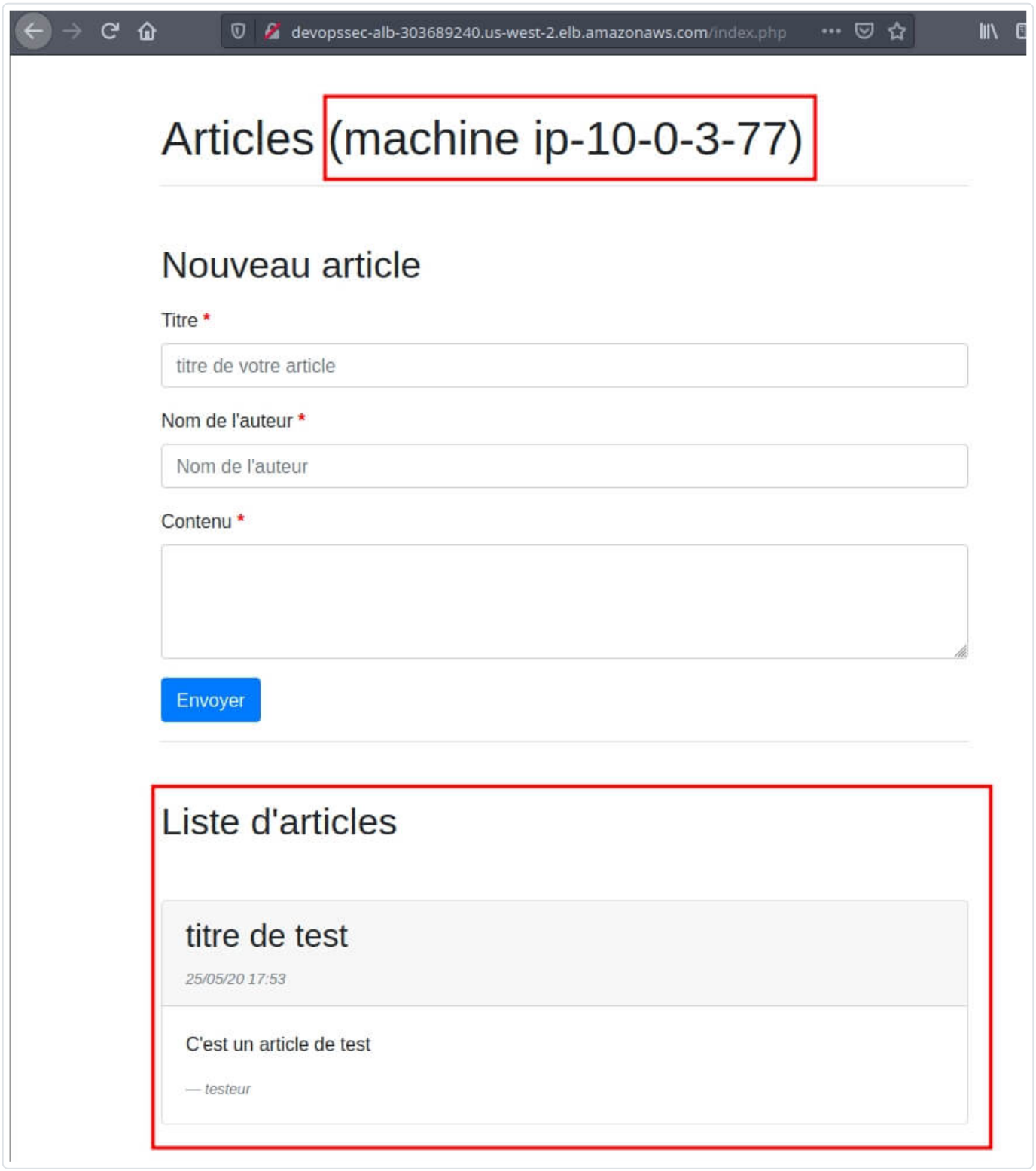




Ensuite je vais rajouter un poste sur mon application web pour vérifier le bon fonctionnement de ma base de données :



Vérifions à nouveau notre page web sur nos deux instances :




devopssec-alb-303689240.us-west-2.elb.amazonaws.com/index.php
Articles (machine ip-10-0-4-157)
Nouveau article
Titre *
titre de votre article
Nom de l'auteur *
Nom de l'auteur
Contenu *
Envoyer
Liste d'articles
titre de test
25/05/20 17:53
C'est un article de test
— testeur

Conclusion

J'espère que vous avez apprécié le projet, et je vous donne rendez-vous dans un prochain chapitre pour découvrir l'outil Packer.