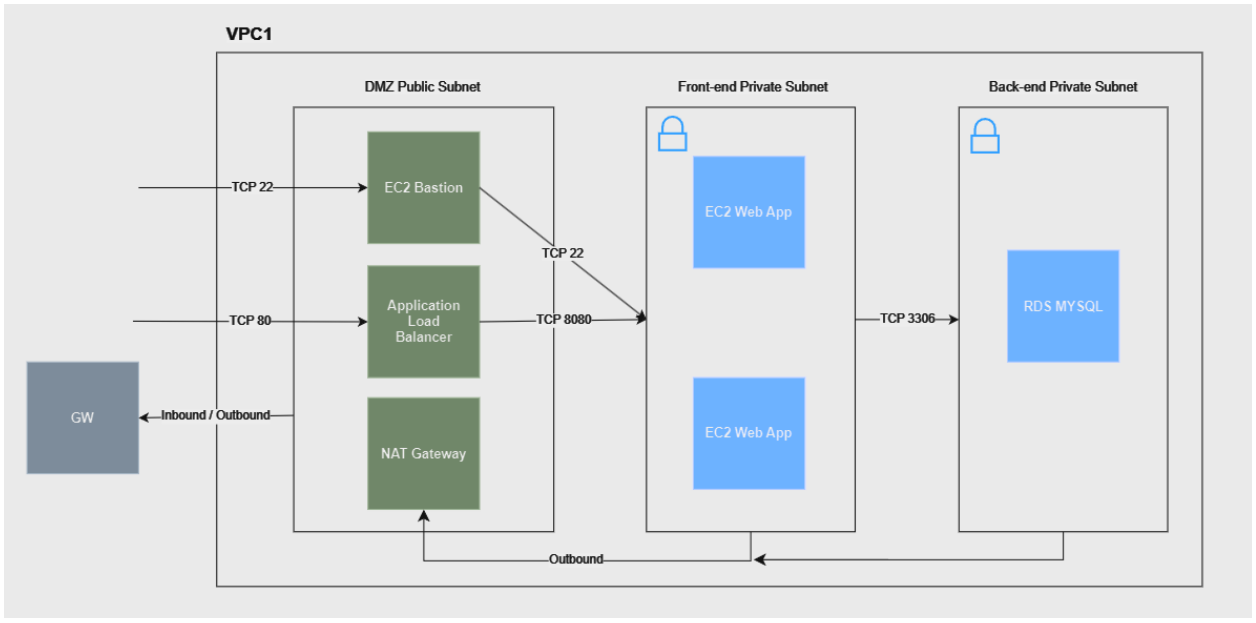
AWS using Terraform

AWS (Amazon Web Services) is one of the pioneers in cloud computing. They provide a huge bunch of cloud services that help startups and big companies. They have diverse services, such as servers (EC2), databases (RDS), domain names (Route53) , load balancers and a lot more. In this report, I will discuss a real-life example of using AWS, but using Terraform, which is an automation tool that allows you to automate resource creation process across all cloud service providers but not only AWS, it supports other cloud computing services such as Microsoft Azure, Google Cloud Platform (GCP) and more. Terraform allows us to create a bunch of resources with a single click, and to destroy all these resources with a single click too. Without Terraform, you would open each resource page on AWS website (or other cloud services providers), and create each resource, then when you want to destroy these resources, you must memorize each one and repeat the same process to destroy them. Let’s start with the example we are going to go through.



We have a single VPC (VPC1) that contains three subnets, the front-end and the back-end are private. Then we have a single public subnet called DMZ (demilitarized zone), this subnet adds an extra layer of protection by isolating the database and the web application from public traffic, they can only be reached by going through the DMZ subnet and its security groups. The DMZ contains three components, the EC2 Bastion is an EC2 instance used to SSH into the front-end web application instances. The Bastion can only be accessed using a private key that is distributed to the employees with the right permission.

Then we have the application load balancer, it distributes the incoming HTTP traffic on port 80 to the web application instances using the round-robin scheduling algorithm. It is a simple algorithm that works exactly like dealing cards to players in a card game, it forwards a request to each instance one by one, more about it later. Finally, we have the NAT gateway which allows the back-end and front-end subnets to access the internet since they need to install some programs and update their systems and so on. What is special about NAT gateways is that they only allow traffic in one direction, in this case, from the two subnets towards the internet, but not vice-versa. The whole VPC is connected to an internet gateway, so it is able to communicate with the outer internet. Now since we talked about the public subnet, we must talk about the private subnets.

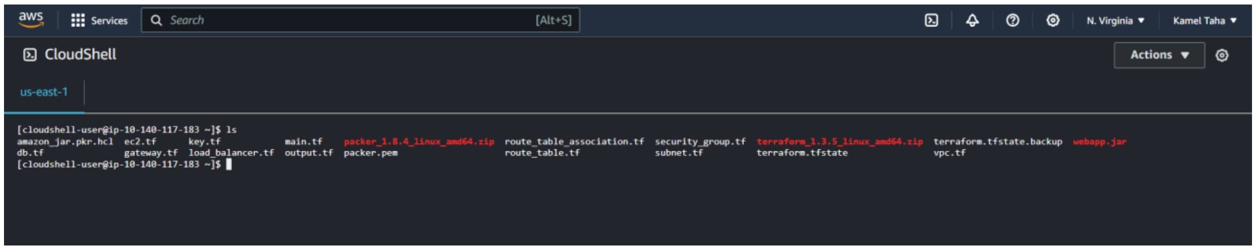
The back-end subnet has an RDS MYSQL database running. RDS is a service provided by AWS for creating and hosting databases, I used the service to create a simple database called “names_db”, the database contains a single table called “names”, the table is created automatically by JPA, which is a Spring dependency used to communicate with databases. The database content is not persistent, once the database is shutdown all data are wiped out. To avoid this, S3 storage might be utilized to persist the data, but I did not implement that.

The front-end subnet has two EC2 instances running our web application, the load balancer distributes the traffic between the two instances. I used Spring framework to create the web application, it is running on port 8080 on each machine. The web application has three simple features, it gets all the names stored inside the names table, and it has a button that generates a random name and stores it in the names table. Finally, the website prints the IP address of the EC2 instance running the web application to differentiate between the two instances.

After discussing the top-down view of the example I want to implement, let’s discuss the implementation.

Working Environment

To implement the example, I used AWS Cloudshell to build all the resources using Terraform. All Terraform files are uploaded to Cloudshell and then applied to create all resources with a single command. The Cloudshell must be setup first with the needed packages, such as installing Terraform and Packer (more about it later). Below we can find all the uploaded and downloaded files in Cloudshell.



```
[cloudshell-user@ip-10-140-117-183 ~]$ ls
amazon_jar.pkr.hcl  ec2.tf      key.tf             main.tf    packer_1.8.4_linux_amd64.zip  route_table_association.tf  security_group.tf  terraform_1.3.5_linux_amd64.zip  terraform.tfstate.backup  webapp.jar
db.tf               gateway.tf  load_balancer.tf   output.tf  packer.pem                    route_table.tf             subnet.tf          terraform.tfstate                vpc.tf
[cloudshell-user@ip-10-140-117-183 ~]$ 
```

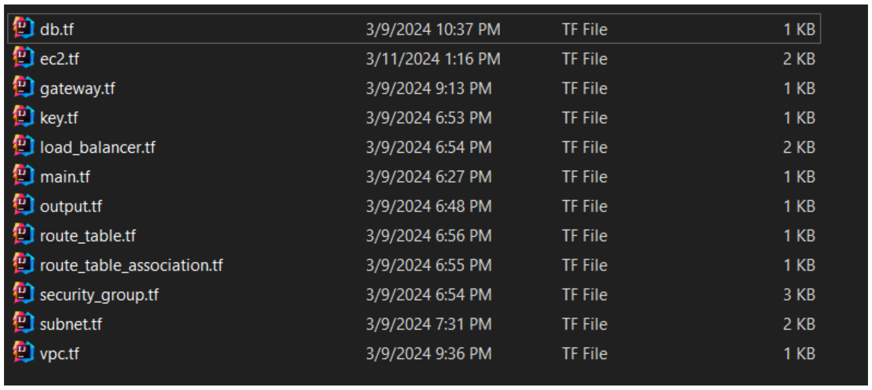
Deep Dive

In this section, I will go through the process of creation for each component in the example. I will go through the process in the following order:

- Terraform setup.
- VPC, internet gateway, and route tables.
- Subnets, security groups, and route table associations.
- RDS.
- Web applications and Packer.
- Bastion setup.
- NAT gateway.
- Load balancer.
- Outputs.
- Test run.
- Future work.
- Conclusion.

Terraform setup

First, let's check the files structure of my Terraform code. We can see from below that common resources are collected in the same file.



db.tf	3/9/2024 10:37 PM	TF File	1 KB
ec2.tf	3/11/2024 1:16 PM	TF File	2 KB
gateway.tf	3/9/2024 9:13 PM	TF File	1 KB
key.tf	3/9/2024 6:53 PM	TF File	1 KB
load_balancer.tf	3/9/2024 6:54 PM	TF File	2 KB
main.tf	3/9/2024 6:27 PM	TF File	1 KB
output.tf	3/9/2024 6:48 PM	TF File	1 KB
route_table.tf	3/9/2024 6:56 PM	TF File	1 KB
route_table_association.tf	3/9/2024 6:55 PM	TF File	1 KB
security_group.tf	3/9/2024 6:54 PM	TF File	3 KB
subnet.tf	3/9/2024 7:31 PM	TF File	2 KB
vpc.tf	3/9/2024 9:36 PM	TF File	1 KB

Then in the code below we can find the code for the main.tf file. I am telling Terraform that we are going to use AWS as a cloud service. Also, I am specifying the AWS region used.

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.16"
    }
  }

  required_version = ">= 1.2.0"
}

provider "aws" {
  region = "us-east-1"
}
```

VPC, internet gateway, and route tables

In the resource below, I am specifying the name of the VPC and the CIDR block used for it, since 16 is the mask for the VPC, then the subnets inside the VPC must have a larger mask like 24.

```
resource "aws_vpc" "VPC1" {
  cidr_block = "10.55.0.0/16"
  tags = {
    Name = "VPC1"
  }
}
```

Then the internet gateway is created to give VPC1 the ability to communicate with the internet.

```
resource "aws_internet_gateway" "gw" {
  vpc_id = aws_vpc.VPC1.id

  tags = {
    Name = "VPC1_GW"
  }
}
```

The VPC has two route tables, one is associated with the public subnet, and another is associated with the private subnets. The public table allows all traffic to go through the gateway in a bi-directional fashion. While the private table connects the private subnets with the NAT gateway to allow them to communicate with the internet in one direction.

```
resource "aws_route_table" "public-routes" {
  vpc_id = aws_vpc.VPC1.id
  route {
    cidr_block = "0.0.0.0/0"
    gateway_id = aws_internet_gateway.gw.id
  }

  tags = {
    Name = "VPC1_PUBLIC_RTBL"
  }
}


resource "aws_route_table" "private-routes" {
  vpc_id = aws_vpc.VPC1.id

  route {
    cidr_block = "0.0.0.0/0"
    nat_gateway_id = aws_nat_gateway.nat-gw.id
  }

  tags = {
    Name = "VPC1_PRIVATE_RTBL"
  }
}
```

Subnets, security groups, and route table associations

All subnets belong to VPC1, this is why we have VPC1 id specified for vpc_id. Then an availability zone is specified to the subnet, and finally, as mentioned before the mask of the CIDR block is 24.

```
resource "aws_subnet" "dmz-subnet" {
  vpc_id            = aws_vpc.VPC1.id
  availability_zone = "us-east-1a"
  cidr_block        = "10.55.10.0/24"
  tags = {
    Name = "dmz-subnet"
  }
}


resource "aws_subnet" "dmz-secondary-subnet" {
  vpc_id            = aws_vpc.VPC1.id
  availability_zone = "us-east-1b"
  cidr_block        = "10.55.11.0/24"
  tags              = {
    Name = "dmz-secondary-subnet"
  }
}
```

We have a secondary DMZ subnet that resides in another availability zone. This subnet is created because the load balancer requires two to have two subnets in different availability zones. Notice the difference between all subnets in the CIDR notation. Both subnets have an association with the route table connected to the internet gateway.

```
resource "aws_route_table_association" "dmz-association" {
  subnet_id      = aws_subnet.dmz-subnet.id
  route_table_id = aws_route_table.public-routes.id
}

resource "aws_route_table_association" "dmz-secondary-association" {
  subnet_id      = aws_subnet.dmz-secondary-subnet.id
  route_table_id = aws_route_table.public-routes.id
}
```

The front-end subnet is used to host the web application instances. It is associated with the private route table that is connected to the NAT gateway.

```
resource "aws_subnet" "front-end-subnet" {
  vpc_id            = aws_vpc.VPC1.id
  availability_zone = "us-east-1a"
  cidr_block        = "10.55.12.0/24"
  tags = {
    Name = "front-end-subnet"
  }
}


resource "aws_route_table_association" "front-end-association" {
  subnet_id      = aws_subnet.front-end-subnet.id
  route_table_id = aws_route_table.private-routes.id
}
```

Then we have the backend subnet, exactly like the load balancer situation, RDS requires to have two subnets each one in a different availability zone.

```
resource "aws_subnet" "back-end-subnet" {
  vpc_id            = aws_vpc.VPC1.id
  availability_zone = "us-east-1a"
  cidr_block        = "10.55.13.0/24"
  tags = {
    Name = "back-end-subnet"
  }
}

resource "aws_subnet" "back-end-secondary-subnet" {
  vpc_id            = aws_vpc.VPC1.id
  availability_zone = "us-east-1b"
  cidr_block        = "10.55.14.0/24"
  tags = {
    Name = "back-end-secondary-subnet"
  }
}
```

Both are associated with the private route table.

```
resource "aws_route_table_association" "back-end-association" {
  subnet_id      = aws_subnet.back-end-subnet.id
  route_table_id = aws_route_table.private-routes.id
}
resource "aws_route_table_association" "back-end-secondary-association" {
  subnet_id      = aws_subnet.back-end-secondary-subnet.id
  route_table_id = aws_route_table.private-routes.id
}
```

Now let us talk about the security groups I have defined. The first group allows the SSH traffic from anywhere. This group is used for the EC2 instance that resides inside the DMZ subnet, also it is used for the EC2 instances inside the front-end subnet.

```
resource "aws_security_group" "allow-ssh" {
  name        = "allow-ssh"
  description = "Allow SSH inbound traffic"
  vpc_id      = aws_vpc.VPC1.id

  ingress {
    description = "SSH to EC2"
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  tags = {
    Name = "allow-ssh"
  }
}
```

The next group allows SQL traffic (port 3306) from anywhere. This group is used for the RDS MYSQL instance inside the back-end subnet.

```
resource "aws_security_group" "allow-sql" {
  name        = "allow-sql"
  description = "Allow SQL inbound traffic"
  vpc_id      = aws_vpc.VPC1.id

  ingress {
    description = "SQL to EC2"
    from_port   = 3306
    to_port     = 3306
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  tags = {
    Name = "allow-sql"
  }
}
```

The below group allows traffic from anywhere through HTTP port 80. The group is used for the load balancer to accept any HTTP traffic.

```
resource "aws_security_group" "allow-http" {
  name        = "allow-http"
  description = "Allow HTTP inbound traffic"
  vpc_id      = aws_vpc.VPC1.id

  ingress {
    description = "HTTP to EC2"
    from_port   = 80
    to_port     = 80
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  tags = {
    Name = "allow-http"
  }
}
```

Then we have the group that allows the traffic through port 8080. The group is used for the EC2 instances that reside inside the front-end subnet.

```
resource "aws_security_group" "allow-8080" {
  name        = "allow-8080"
  description = "Allow Web inbound traffic"
  vpc_id      = aws_vpc.VPC1.id

  ingress {
    description = "Web to EC2"
    from_port   = 8080
    to_port     = 8080
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  tags = {
    Name = "allow-8080"
  }
}
```

Then the group that allows the ping traffic. This group is used for testing purposes only.

```
resource "aws_security_group" "allow-ping" {
  name        = "allow-ping"
  description = "Allow ping"
  vpc_id      = aws_vpc.VPC1.id

  ingress {
    description = "Ping"
    from_port   = -1
    to_port     = -1
    protocol    = "icmp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  tags = {
    Name = "allow-ping"
  }
}
```

Finally, the group that allows all egress traffic through all ports. This group is used to allow private instances to access anything from the internet.

```
resource "aws_security_group" "allow-all-outbound" {
  name        = "allow-all-outbound"
  description = "Allow all outbound traffic"
  vpc_id      = aws_vpc.VPC1.id

  egress {
    description = "All outbound"
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }

  tags = {
    Name = "allow-all-outbound"
  }
}
```

RDS

To create a database using RDS, you must create a subnet group to run the database inside. The group must contain two subnets at least, each one must reside in a different availability zone. The database storage is 20 GB, it runs MYSQL and has a default database called “names_db”. The username and password of the database are specified below. Then finally, we specify the security groups for the database. The RDS is a very expensive component, it takes 3-5 minutes to start.

```
resource "aws_db_subnet_group" "backend-db-subnet-group" {
  name       = "main"
  subnet_ids = [aws_subnet.back-end-subnet.id, aws_subnet.back-end-secondary-
subnet.id]

  tags = {
    Name = "Backend DB subnet group"
  }
}

resource "aws_db_instance" "back-end-db" {
  allocated_storage         = 20
  storage_type              = "gp2"
  engine                   = "mysql"
  engine_version           = "5.7"
  instance_class           = "db.t3.micro"
  db_name                  = "names_db"
  username                 = "user"
  password                 = "password"
  parameter_group_name     = "default.mysql5.7"
  db_subnet_group_name     = aws_db_subnet_group.backend-db-subnet-group.id
  vpc_security_group_ids   =  [
    aws_security_group.allow-sql.id,
    aws_security_group.allow-all-outbound.id,
  ]
  skip_final_snapshot      = true
}
```

Web applications and Packer

I created my web application using Spring framework. The features of the application are mentioned previously. Here I will talk about the implementation itself. The application has three APIs as follows:

“/”: used to get all the names from the database.

“/name”: used to add a name to the database.

“/health”: used by the load balancer to check the health of the instance.

For communication with the database, I have used JPA which allows you to create an annotation-based configuration in a very quick manner, that suits my simple website. ThymeLeaf is used to create a single webpage, it takes the list of names and displays them.

Finally, I have the application.properties file with the configuration below.

```
spring.datasource.url=jdbc:mysql://${DB_HOST}:${DB_PORT:3306}/${DB_NAME:names_db}
spring.datasource.username=${DB_USER:user}
spring.datasource.password=${DB_PASSWORD:password}
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.jpa.database-platform=org.hibernate.dialect.MySQLDialect
spring.jpa.hibernate.ddl-auto=update
```

The project is packaged into a JAR file. When running this JAR file, you must specify the hostname of the database at least, you can play with other arguments such as the port, the database name, the username, and the password, but they are optional. Below we can see how we can run our web application:

```
nohup java -jar webapp.jar --DB_HOST=${aws_db_instance.back-end-db.address
```

We use “nohup” to run the app in the background without displaying the logs.

But wait, if we are building our EC2 instances with an AMI that is provided by AWS, then we will face a problem! We try to run the JAR file, but the file does not exist in the image by default. Let’s see how I solved this problem!

Packer is a software created by the same company that created Terraform, which is “HashiCorp”. This software allows you to build your own custom AWS images based on an existing image. This image is stored in your account in AWS with a specific AMI, then you can use this AMI to create EC2 instances. So, what I did is that I took Amazon Linux’s AMI as a base image and created an image with the JAR file pre-installed. Below we can find the code explanation.

```
source "amazon-ebs" "amazon_linux" {
  ami_name                = "my-amazon-linux-ami"
  source_ami              = "ami-0f403e3180720dd7e"
  instance_type           = "t2.micro"
  region                  = "us-east-1"
  ssh_username            = "ec2-user"
  ssh_keypair_name        = "packer"
  ssh_private_key_file    = "packer.pem"
}

build {
  sources        = ["source.amazon-ebs.amazon_linux"]
  provisioner "file" {
    source      = "webapp.jar"
    destination = "webapp.jar"
  }
  provisioner "shell" {
    inline = [
      "sudo yum update -y",
      "sudo dnf install java-17-amazon-corretto -y"
    ]
  }
}
```

In the source block we specify based on what image we want to build our custom one. All the needed details to create an EC2 instance are specified, it looks very much like an EC2 template. In the same block, we assign a key so that packer can communicate with the image through SSH and install out JAR file or execute any needed commands on new image’s shell. Then we have the build block, it uses the pre-defined source, in addition to any specified provisioners. The file provisioner copies a file from a source to a destination, for that purpose we must have our “webapp.jar” file in the same directory as the packer file, that is why the JAR file is uploaded to Cloudshell. Then we have the shell provisioner that executes the given list of commands, the commands update the packages, and installs java on that image. All these steps must be done so this line from the user data section works correctly.

But are there any other solutions to solve this problem? Yes, the most simple one is to SSH into each web application instance and install the JAR manually, but this is not scalable!

```
nohup java -jar /home/ec2-user/webapp.jar
--DB_HOST=${aws_db_instance.back-end-db.address}
--DB_PORT=${aws_db_instance.back-end-db.port}
--DB_USER=user
--DB_PASSWORD=password
--DB_NAME=names_db
> /home/ec2-user/output.log
```

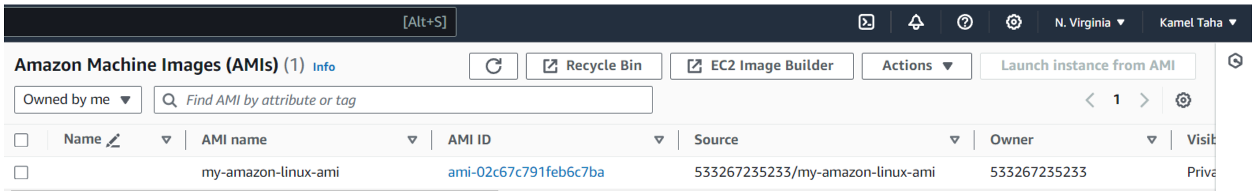
After creating the AMI image, we can use it in our “web-server” EC2 resource. The count attribute means that we want to create 2 instances of this resource. We can use the count later in the block as an index to specify in which loop we are operating. It depends on the key pair resource that will be mentioned below, also it depends on the back-end instance, which is the database. And as we can see, we use the AMI of the image we created using Packer. Finally, the security groups allow inbound SSH, 8080 and ping traffic, and they allow all outbound traffic.

```
resource "aws_instance" "web-server" {
  count                       = 2
  depends_on                  = [aws_key_pair.ec2-key-pair, aws_db_instance.back-end-
db]
  subnet_id                   = aws_subnet.front-end-subnet.id
  ami                         = "ami-02c67c791feb6c7ba" # AMI of my custom amazon
image
  instance_type               = "t2.micro"
  associate_public_ip_address = false
  key_name                    = "kamel_key_pair"
  vpc_security_group_ids      = [
    aws_security_group.allow-ssh.id,
    aws_security_group.allow-8080.id,
    aws_security_group.allow-all-outbound.id,
    aws_security_group.allow-ping.id
  ]

  user_data = # Omitted code, you can find the code in the assignment files.


  tags = {
    Name ="instance-${count.index}"
  }
}
```

Here we can find the resource for the mentioned key:

```
resource "aws_key_pair" "ec2-key-pair" {
  key_name   = "kamel_key_pair"
  public_key = tls_private_key.rsa.public_key_openssh
}

resource "tls_private_key" "rsa" {
  algorithm = "RSA"
  rsa_bits  = 4096
}

resource "local_file" "ec2-key" {
  content  = tls_private_key.rsa.private_key_pem
  filename = "kamel"
}
```

And the custom image on AWS:



Name	AMI name	AMI ID	Source	Owner	Visib
	my-amazon-linux-ami	ami-02c67c791feb6c7ba	533267235233/my-amazon-linux-ami	533267235233	Priva

Bastion setup

Instances that stand as a firewall preventing any traffic from reaching our resources are called Bastion. I have a single Bastion in my implementation that we can SSH into. Then through it we can SSH into the front-end instances. It uses the default Amazon Linux image allowing inbound SSH and all outbound traffic.

```
resource "aws_instance" "bastion-server" {
  depends_on                  = [aws_key_pair.ec2-key-pair]
  subnet_id                   = aws_subnet.dmz-subnet.id
  ami                         = "ami-0f403e3180720dd7e"
  instance_type               = "t2.micro"
  associate_public_ip_address = true
  key_name                    = "kamel_key_pair"
  vpc_security_group_ids      = [aws_security_group.allow-ssh.id,
aws_security_group.allow-all-outbound.id]

  tags = {
    Name     = "Bastion Server"
  }
}
```

NAT gateway

A NAT gateway is needed for duty, so the components in private subnets can reach out for the internet, without letting the internet reach the components, it works only in one direction. But why would we need such a thing? Simply we can say because of this line in the user data of the web application EC2 instances.

```
yum update -y
```

The update command needs to reach the internet to get all the new packages, without a NAT gateway, it is not possible. Below we can see how the NAT gateway is created using Terraform.

First, we create an elastic IP address because it is required to create a NAT gateway.

```
resource "aws_eip" "nat_eip" {
  vpc = true
}
```

Then we create the NAT gateway itself, we must put it inside the public DMZ subnet because this subnet is linked to the internet gateway. So, the private subnets send traffic to the NAT, then the NAT sends the traffic to the internet gateway. That's why I am specifying a depends-on condition on the NAT gateway.

```
resource "aws_nat_gateway" "nat-gw" {
  subnet_id = aws_subnet.dmz-subnet.id
  allocation_id = aws_eip.nat_eip.id

  depends_on = [aws_internet_gateway.gw]
}
```

Load balancer

A load balancer is required when you have a very high load, for a high load you need to use multiple servers (EC2 instances) to handle all the requests. At the same time, you need some sort of originating for the process of forwarding the requests to be handled by a specific instance. Let's see how it is implemented using Terraform.

First, we need something called a target group. A target group tells the load balancer where to send the requests, what are the instances contributing to handle these requests?

```
resource "aws_lb_target_group" "target-group" {
  name        = "nit-tg"
  port        = 8080
  protocol    = "HTTP"
  target_type = "instance"
  vpc_id      = aws_vpc.VPC1.id

  health_check {
    enabled             = true
    interval            = 10
    path                = "/health"
    port                = "traffic-port"
    protocol            = "HTTP"
    timeout             = 6
    healthy_threshold   = 2
    unhealthy_threshold = 2
  }
}
```

We tell the load balancer to use this target group, and send requests to port 8080 on these instances, because our Spring application is running on that port. Also, a health check is defined so the load balancer can decide whether a specific instance is operating correctly or not. I have a "/health" API defined in my Spring application so that the check can send a request to that endpoint and get a 200-response identifying that the instance is working as expected.

Then we have the load balancer resource itself.

```
resource "aws_lb" "application-lb" {
  name               = "nit-alb"
  depends_on         = [aws_instance.web-server]
  internal           = false
  load_balancer_type = "application"
  subnets            = [aws_subnet.dmz-subnet.id, aws_subnet.dmz-secondary-subnet.id]
  security_groups    = [aws_security_group.allow-http.id, aws_security_group.allow-
all-outbound.id]
  ip_address_type    = "ipv4"

  tags = {
    name = "nit-alb"
  }
}
```

It depends on the EC2 instances, so the instance must run first to get the load balancer working correctly. And we are specifying that it is an application load balancer since this type is suitable for application layer applications. And we are choosing two subnets from different availability zones, this is required for the balancer so if something wrong happens in one availability zone, we have another correctly-operating one. Also, we are defining our security groups by allowing inbound HTTP requests and allowing all outbound traffic.

Finally, we have two more resources. A listener is required for the load balancer to receive traffic on HTTP port 80 before forwarding it to the instances, the listener is connected to both the target group and the load balancer as we can see. Then we have several target group attachments. These attachments are defined to tell the target group how to find its targets. So, we are creating as many attachments as the number of working instances, defined by "web-server" list.

```
resource "aws_lb_listener" "alb-listener" {
  load_balancer_arn = aws_lb.application-lb.arn
  port              = 80
  protocol          = "HTTP"

  default_action {
    type             = "forward"
    target_group_arn = aws_lb_target_group.target-group.arn
  }
}

resource "aws_lb_target_group_attachment" "ec2-web-attachment" {
  count            = length(aws_instance.web-server)
  target_group_arn = aws_lb_target_group.target-group.arn
  target_id        = aws_instance.web-server[count.index].id
}
```

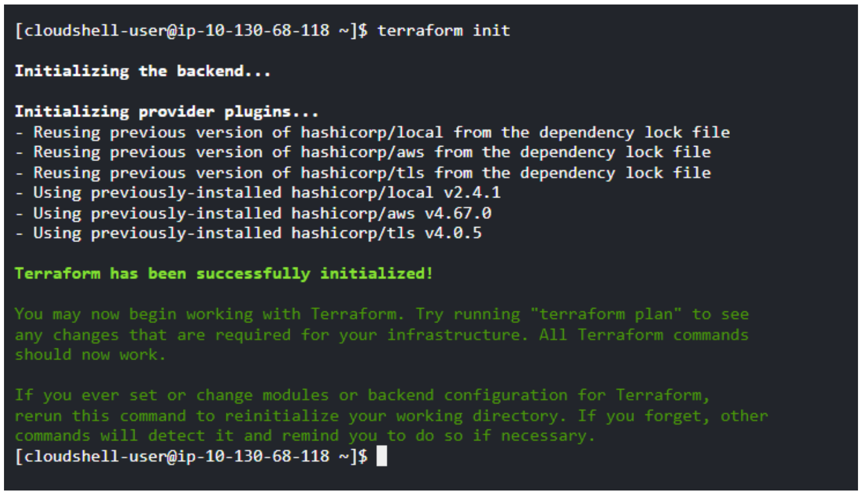
Outputs

This section is very short, it only prints the public IP address of the Bastion so we can SSH into it without referring to AWS dashboard. Also, it prints the private IP addresses of the web application instances so we can SSH into them, and finally, the address of the load balancer which is the end point that clients will use to reach our service.

```
output "bastion-public-ip" {
  value = aws_instance.bastion-server.public_ip
}

output "web-apps-private-ips" {
  value = aws_instance.web-server.*.private_ip
}
output "elb-dns-name" {
  value = aws_lb.application-lb.dns_name
}
```

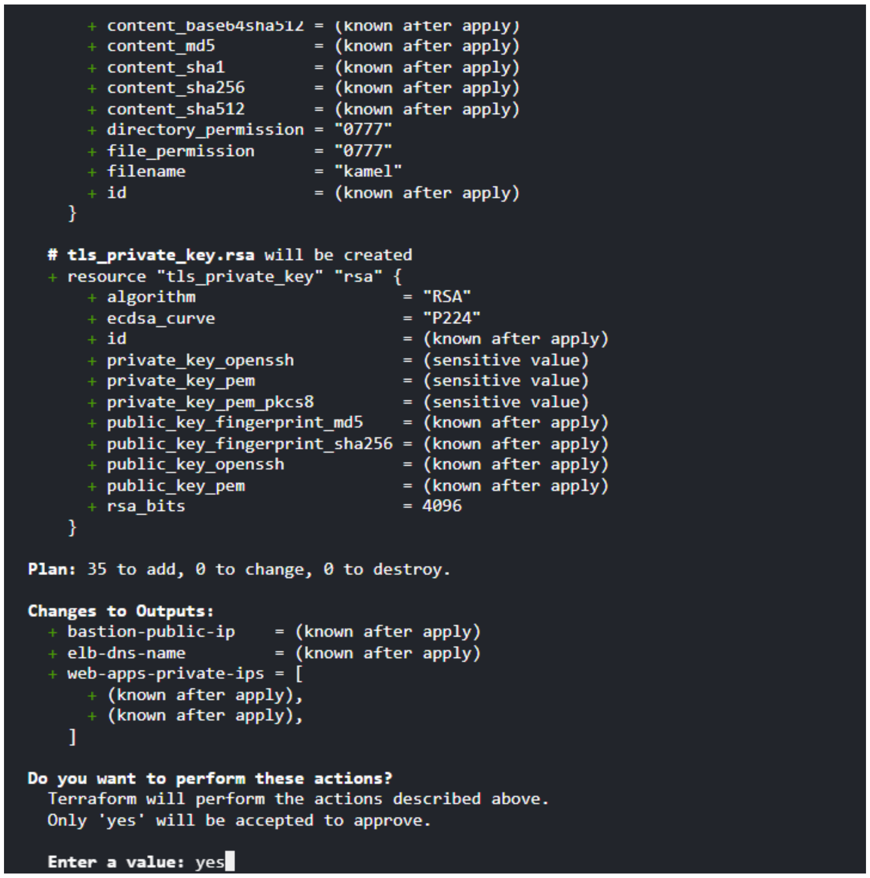
Test run

First, we run “terraform init” in the same directory where terraform files reside.



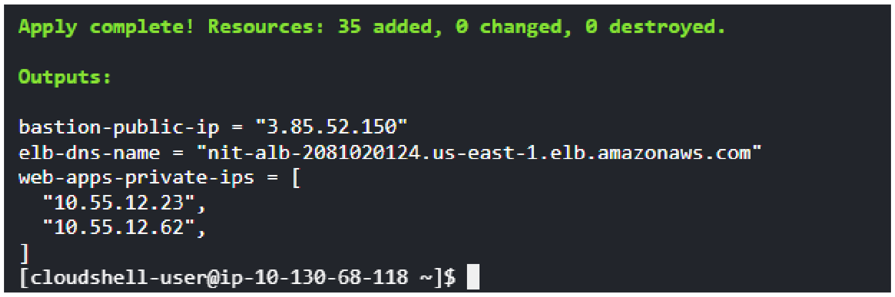
```
[cloudshell-user@ip-10-130-68-118 ~]$ terraform init

Initializing the backend...

Initializing provider plugins...
- Reusing previous version of hashicorp/local from the dependency lock file
- Reusing previous version of hashicorp/aws from the dependency lock file
- Reusing previous version of hashicorp/tls from the dependency lock file
- Using previously-installed hashicorp/local v2.4.1
- Using previously-installed hashicorp/aws v4.67.0
- Using previously-installed hashicorp/tls v4.0.5

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
[cloudshell-user@ip-10-130-68-118 ~]$
```

Then we run “terraform apply” and enter yes to create all resources.



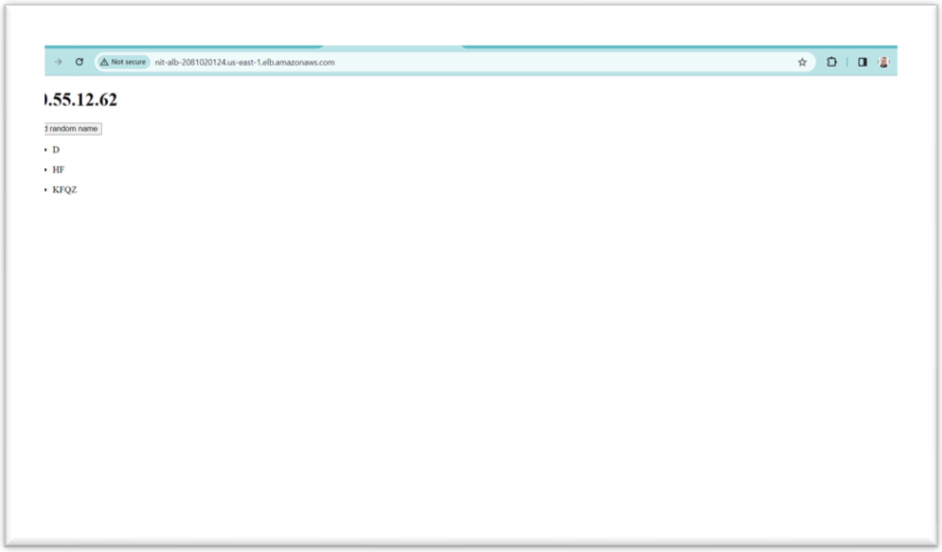
```
      + content_base64sha512 = (known after apply)
      + content_md5          = (known after apply)
      + content_sha1         = (known after apply)
      + content_sha256       = (known after apply)
      + content_sha512       = (known after apply)
      + directory_permission = "0777"
      + file_permission      = "0777"
      + filename             = "kamel"
      + id                   = (known after apply)
    }

  # tls_private_key.rsa will be created
  + resource "tls_private_key" "rsa" {
      + algorithm                     = "RSA"
      + ecdsa_curve                   = "P224"
      + id                            = (known after apply)
      + private_key_openssh           = (sensitive value)
      + private_key_pem               = (sensitive value)
      + private_key_pem_pkcs8         = (sensitive value)
      + public_key_fingerprint_md5    = (known after apply)
      + public_key_fingerprint_sha256 = (known after apply)
      + public_key_openssh            = (known after apply)
      + public_key_pem                = (known after apply)
      + rsa_bits                      = 4096
    }

Plan: 35 to add, 0 to change, 0 to destroy.

Changes to Outputs:
  + bastion-public-ip     = (known after apply)
  + elb-dns-name          = (known after apply)
  + web-apps-private-ips = [
      + (known after apply),
      + (known after apply),
    ]

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes
```

After several minutes we get the outputs, and we can find all the resources running on AWS dashboard.



```
Apply complete! Resources: 35 added, 0 changed, 0 destroyed.

Outputs:

bastion-public-ip = "3.85.52.150"
elb-dns-name = "nit-alb-2081020124.us-east-1.elb.amazonaws.com"
web-apps-private-ips = [
  "10.55.12.23",
  "10.55.12.62",
]
[cloudshell-user@ip-10-130-68-118 ~]$ 
```

And we can see how the load balancer distributes the requests on all instances, and how the instances are getting the same data from the database.

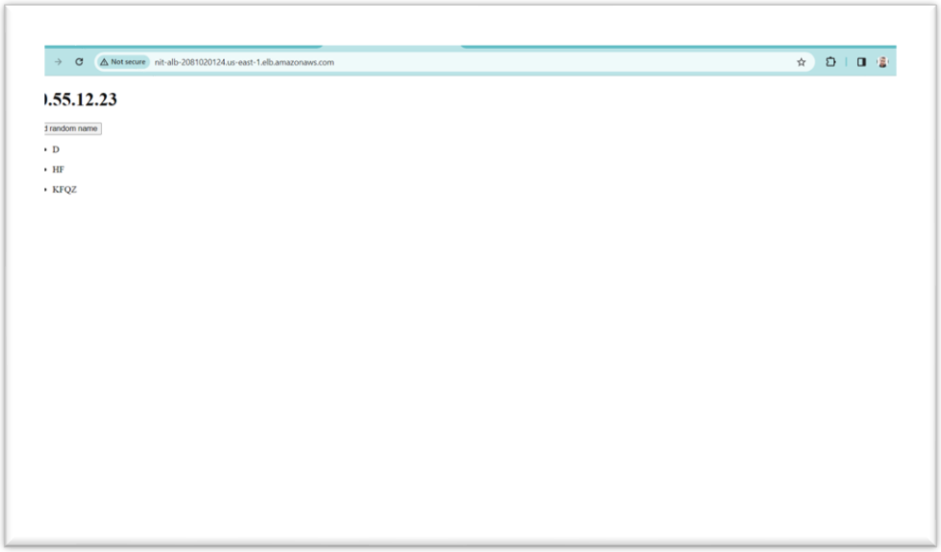


Not secure nit-alb-2081020124.us-east-1.elb.amazonaws.com

).55.12.62

d random name

- D
- HF
- KFQZ



Not secure nit-alb-2081020124.us-east-1.elb.amazonaws.com

).55.12.23

d random name

- D
- HF
- KFQZ

Future work

- I have a single concern about the web application. Every edit to that web application requires rebuilding the AMI image again using Packer, which is not very friendly. I think a CI/CD pipeline might solve this problem by triggering it on each push to the GitHub repository.
- The Spring web application might be dockerized before being added to the instance.

Conclusion

We can see how Terraform makes our life easier by automating the process of creating cloud computing resources. It saves a lot of time and effort. A couple of years ago I used AWS for some homework in a university course. When I was done with the homework, I forgot to destroy some resource and I was charged for it. This is not a possible scenario with Terraform since you can destroy all resources with a single command from the terminal.