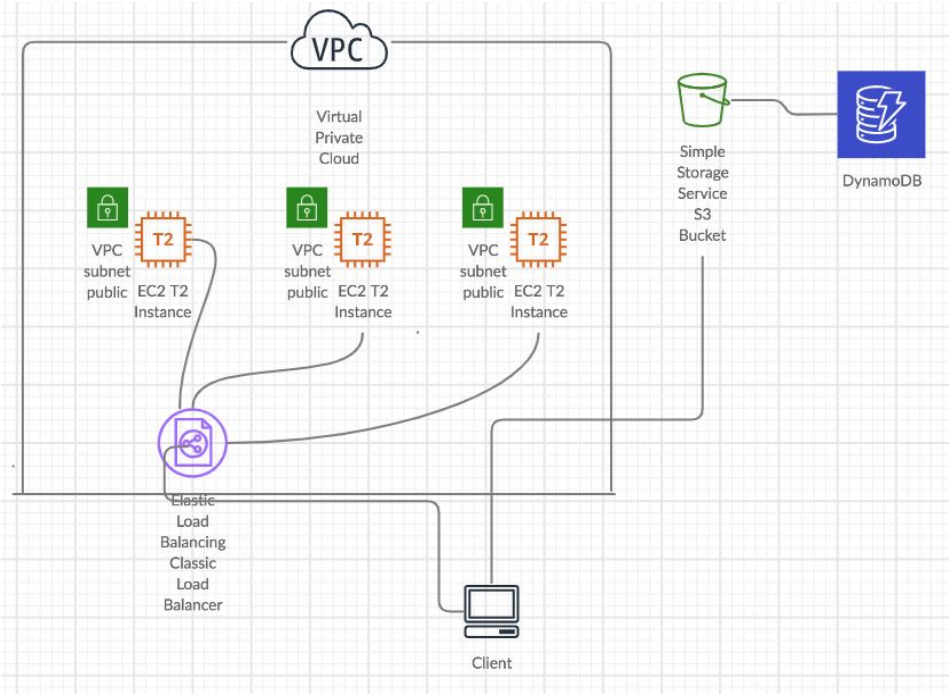
Bink Infrastructure Test

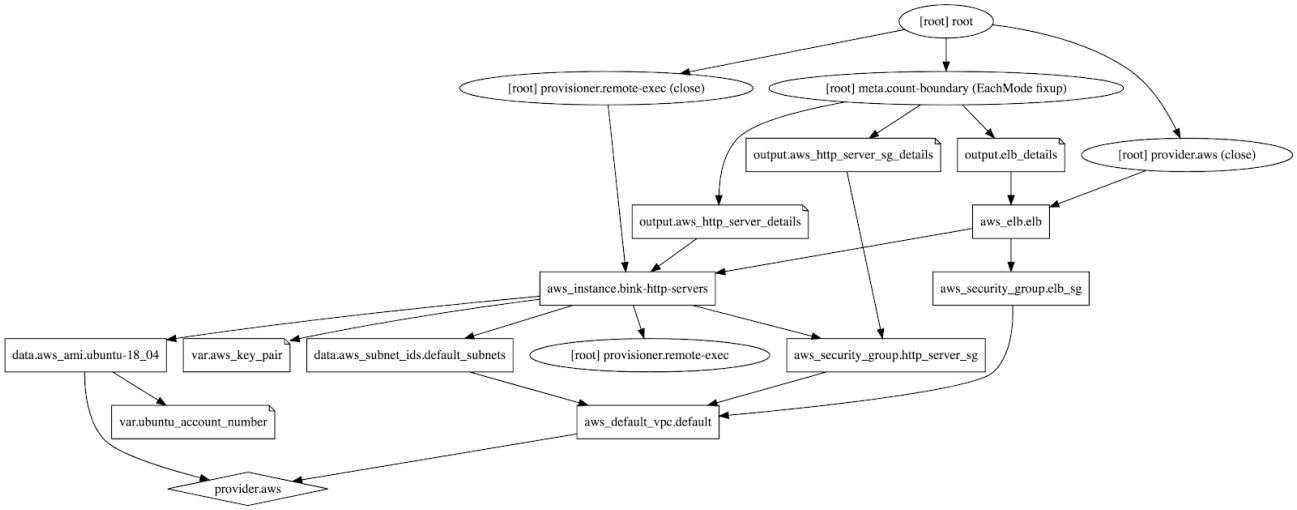
Infrastructure Test

Build out some basic Infrastructure for AWS, Azure or GCP using tooling like Terraform that can be used in a repeatable way. Bonus points for the following:

- Clearly explaining why you're doing things a certain way.
- Providing a PNG diagram of your infrastructure.

Infrastructure Diagrams





Walkthrough

1. Installed Terraform
2. Added a **provider** block to enable the use of AWS resources

```
main.tf > ...
1  provider "aws" {
2    region = "eu-west-2"
3  }
```

3. **terraform init** to download the provider to my local machine

```
→  03_infrastructure_test terraform init

Initializing the backend...

Initializing provider plugins...
- Checking for available provider plugins...
- Downloading plugin for provider "aws" (hashicorp/aws) 3.5.0...
```

4. Created an AWS user to get access keys. This allows Terraform to communicate with AWS.

		User	Access key ID	Secret access key
▸	✓	terraform-user	AKIAT6H2PHPLWSLWAIWV	********* Show

5. Used environment variables to avoid hard coding the access keys information in the terraform script. Terraform scripts will be pushed to a SCM therefore will be seen by the public. These access keys are important and can be misused if exposed to the public.
    - Make sure to delete the user if credentials were to accidentally be exposed.

```
→  03_infrastructure_test export AWS_ACCESS_KEY_ID=AKIAT6H2PHPLWSLWAIWV
→  03_infrastructure_test export AWS_SECRET_ACCESS_KEY=neQdbTlTYV378Y1gWqRvtChAQDKpy52NW9iRBsTD
```

6. Next, I wanted to create an HTTP server
    - Users usually access an HTTP server through Port 80 or Port 443
    - We also might want to SSH into it using Port 22
    - Since this is a web server, we allow HTTP traffic from all IP addresses. We use CIDR blocks to specify IP ranges (0.0.0.0/0)

7. The first step was to create a security group which will be associated with the HTTP server instances.
    - **Ingress** - rules for inbound traffic
    - **Egress** - rules for outbound traffic. Security groups in AWS are stateful which means any changes applied to an inbound rule will automatically be applied to an outbound rule. However, Terraform disables the egress by default so we have to allow this explicitly.

```
resource "aws_security_group" "http_server_sg" {
  name   = "http_server_sg"
  vpc_id = "vpc-2b377243"
```

```
  ingress {
    from_port   = 80
    to_port     = 80
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
  ingress {
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
```

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

8. I created a key pair which will be used by Terraform to perform remote execution on the HTTP server.
    - I stored the .pem file in the **~/aws/aws_keys** directory
    - Changed file permissions

```
➜  aws_keys mv ~/Downloads/bink-keypair.pem .
```

```
➜  aws_keys chmod 400 bink-keypair.pem
```

9. Next, I created the EC2 instance for the HTTP server

```
resource "aws_instance" "bink-http-server" {
  ami                    = "ami-0287acb18b6d8efff" // Ubuntu image
  key_name               = "bink-keypair"
  instance_type          = "t2.micro"
  vpc_security_group_ids = [aws_security_group.http_server_sg.id] /
  subnet_id              = "subnet-fe470897"
}
```

10. SSH into EC2 instance by using the **connection** block
    - **self. public_ip** allows us to refer to the public IP address of the EC2 instance since the **connection** block is within the **resource** block

```
connection {
  type = "ssh"
  host = self.public_ip
  user = "ubuntu"
  private_key = file(var.aws_key_pair)
}
```

11. Created a variable for the aws key pair

```
variable "aws_key_pair" {
  default = "~/aws/aws_keys/bink-keypair.pem"
}
```

12. Next, I used the **remote-exec** provisioner to install Apache in the EC2 instance
    - **Note:** Most IaC is implemented as **immutable infrastructure** - infrastructure that can't be modified once originally provisioned. If immutable infrastructure needs to be changed, it has to be replaced with new infrastructure.
    - Basically, do **terraform destroy** then **terraform apply** again if infrastructure was already provisioned before installing Apache

```
provisioner "remote-exec" {
  inline = [
    "sudo apt update",
    "sudo apt install apache2 -y",
    "sudo systemctl start apache2.service",
    "sudo systemctl enable apache2.service",
    "echo Welcome to Bink - This server is at ${self.public_dns} | sudo tee /var/www/html/index.html"
  ]
}
```

Welcome to Bink - This server is at ec2-18-132-205-54.eu-west-2.compute.amazonaws.com

13. I used the **aws_deafult_vpc** resource to remove hard coding of the default VPC of a specific region. This resource will not create nor destroy a VPC, it will simply "adopt" it into our infrastructure. Overall, this will make the **vpc_id** dynamic.

```
 8   resource "aws_default_vpc" "default" {
 9
10   }
```

- Now if any other resource requires the **vpc_id**, it can just be referred to like this:

```
16    vpc_id = aws_default_vpc.default.id // Which VP
```

14. Next, I also wanted to make the subnets dynamic by using data providers

```
12  data "aws_subnet_ids" "default_subnets" {
13    vpc_id = aws_default_vpc.default.id
14  }
```

- Now, the **subnet_id** can be referred to like below
- The **tolist** function was used to be able to point to a specific subnet. The subnet ids are stored as a set therefore must be converted to a list first.

```
resource "aws_instance" "bink-http-server" {
  ami                    = "ami-0287acb18b6d8efff" // Ubuntu image
  key_name               = "bink-keypair"
  instance_type          = "t2.micro"
  vpc_security_group_ids = [aws_security_group.http_server_sg.id] // Refer to
//subnet_id              =  "subnet-fe470897"
  subnet_id              = tolist(data.aws_subnet_ids.default_subnets.ids)[2]
```

15. I also made the AMI dynamic using another data provider. This will get the latest Ubuntu 18.04 image.

```
22  data "aws_ami" "ubuntu-18_04" {
23    most_recent = true
24    owners = ["${var.ubuntu_account_number}"]
25    filter {
26      name = "name"
27      values = ["ubuntu/images/hvm-ssd/ubuntu-bionic-18.04-amd64-server-*"]
28    }
29  }
```

- Then I created a variable for the Ubuntu's owner account number

```
5   // Canonical number
6   variable "ubuntu_account_number" {
7     default = "099720109477"
8   }
```

- After **terraform apply**, find the data using **terraform console**

```
> data.aws_ami.ubuntu-18_04.id
ami-0287acb18b6d8efff
```

- The AMI ID provided is indeed the AMI ID I hard coded before

```
63  resource "aws_instance" "bink-http-server" {
64    #ami                    = "ami-0287acb18b6d8efff" // Ubuntu image
65    ami                    = data.aws_ami.ubuntu-18_04.id
```

16. At this point, my **main.tf** file was getting cluttered
    - I created a separate file for variables and data providers

17. Next, I wanted to add a load balancer in my infrastructure
    - I created one instance for each subnet in the region
    - To do this, I modified my http servers resource and added a **for_each** to loop through all the subnets within the default VPC in the region

```
43  /* HTTP Server */
44  resource "aws_instance" "bink-http-servers" {
45    #ami                    = "ami-0287acb18b6d8efff" // Ubuntu image
46    ami                    = data.aws_ami.ubuntu-18_04.id
47    key_name               = "bink-keypair"
48    instance_type          = "t2.micro"
49    vpc_security_group_ids = [aws_security_group.http_server_sg.id] //
50
51    for_each  = data.aws_subnet_ids.default_subnets.ids
52    subnet_id = each.value
```

    - I created a security group for the load balancer. The security group only allows HTTP traffic in and allow all traffic out

```
44  resource "aws_security_group" "elb_sg" {
45    name   = "elb_sg"
46    vpc_id = aws_default_vpc.default.id
```

    - I created the actual load balancer, which listens for requests at Port 80. The **instances** were left blank for now. This will be for the list of all the instances our load balancer will redirect traffic to.

```
63  resource "aws_elb" "elb" {
64    name = "bink-elb"
65    subnets = aws_subnet_ids.default_subnets.ids
66    security_groups = [aws_security_group.elb_sg.id]
67    instances = ""
68
69    listener {
70      instance_port = 80
71      instance_protocol = http
72      lb_port = 80
73      lb_protocl = http
74    }
75  }
```

    - To get the list of instances, I used **terraform console**
    - The problem is that the instances will be outputted as maps

```
> aws_instance.bink-http-servers
{
  "subnet-593be515" = {
    "ami" = "ami-0287acb18b6d8efff"
    "arn" = "arn:aws:ec2:eu-west-2:271108226007:instance/i-0bf7b0d6300aa0cf6"
    "associate_public_ip_address" = true
    "availability_zone" = "eu-west-2b"
    "cpu_core_count" = 1
    "cpu_threads_per_core" = 1
    "credit_specification" = [
      {
        "cpu_credits" = "standard"
      },
```

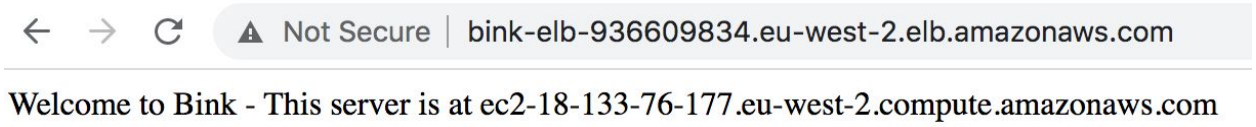
- In order load balance between the instances, I needed to get the id of each http server
- For this, I used the **values** function to convert a map to a set format. Then from this, the ids can be picked up

```
> values(aws_instance.bink-http-servers).*.id
[
  "i-0bf7b0d6300aa0cf6",
  "i-01b88a2b72f249018",
  "i-0afead51c60158551",
]
```

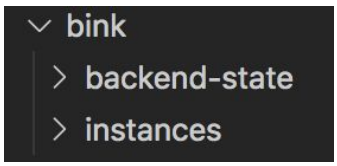
- Now the load balancer should be able to distribute traffic to all the instances

```
67  instances = values(aws_instance.bink-http-servers).*.id
```

18. Checked the load balancer public dns and it does redirect traffic to the http servers



19. Next, I created a remote backend to store my Terraform state
    - I used AWS S3 as my remote backend to store my Terraform state
    - DynamoDB for locking. This ensures that only one user at any point in time can update the state and be able to execute anything on the environment
    - To do this, I created a new project which will deal with the remote backend. Then I put all of the instances' infrastructure in a separate project. Overall, my folder structure looked like this



- Then I created the s3 bucket

```
resource "aws_s3_bucket" "bink_backend_state" {
  bucket = "dev-bink-backend-state"
  // Prevents deletion of resource if there's an accidental terraform destroy
  lifecycle {
    prevent_destroy = true
  }
  // Enables versioning
  versioning {
    enabled = true
  }
  // Server Side Encryption
  server_side_encryption_configuration {
    rule {
      apply_server_side_encryption_by_default {
        sse_algorithm = "AES256"
      }
```

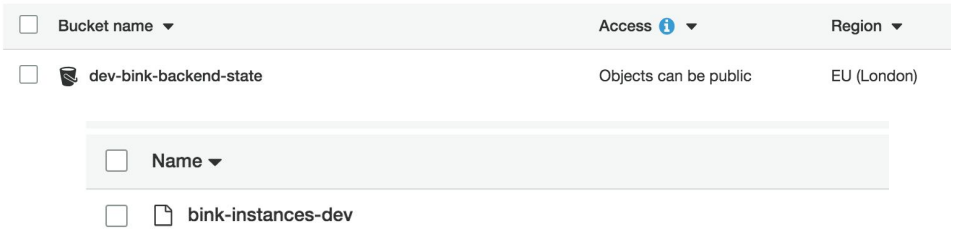
- I also added the DynamoDB

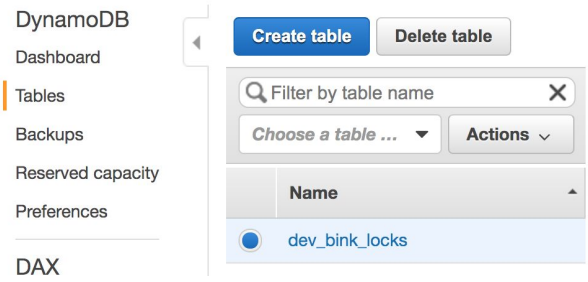
```
// Locking - DynamoDB
resource "aws_dynamodb_table" "bink-backend-lock" {
  name         = "dev_bink_locks"
  billing_mode = "PAY_PER_REQUEST"

  hash_key = "LockID"
  attribute {
    name = "LockID"
    type = "S"
  }
}
```

- Lastly, in the **main.tf** file of **instances** project, I modified it to connect to the remote backend.

```
terraform {
  backend "s3" {
    bucket = "dev-bink-backend-state"
    key = "bink-instances-dev"
    region = "eu-west-2"
    dynamodb_table = "dev_bink_locks"
    encrypt = true
  }
```

20. Finally, I checked if the state were in fact stored in the S3 bucket and if DynamoDB was also present.





Notes:

terraform console can be used to inspect resources. The details can be used to define what we can put inside the output variables

terraform validate to check for syntax errors

terraform fmt to format your scripts in case a formatter is not available