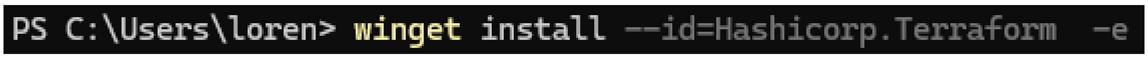
Terraform Workshop

GitHub Link: /terraform-workshop

Part 1: Terraform Basics

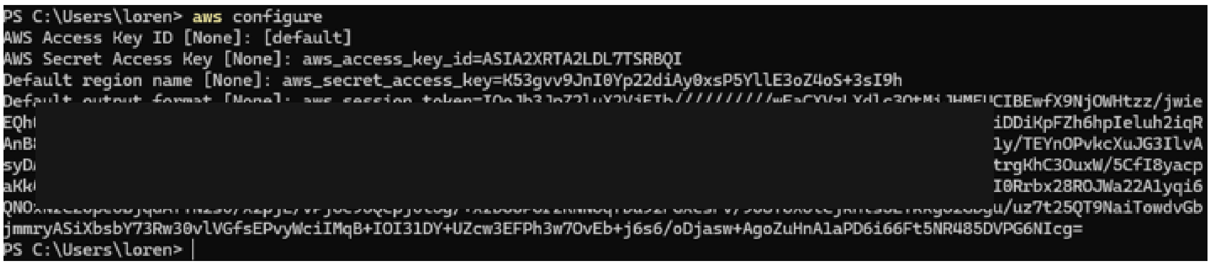
Configuring Terraform

Our first step was to install terraform.



```
PS C:\Users\loren> winget install --id=Hashicorp.Terraform  -e
```

Next, we had to configure our AWS credentials, to allow terraform to access our aws lab. Having already installed AWS CLI, we simply ran *aws configure* and pasted our aws access key.

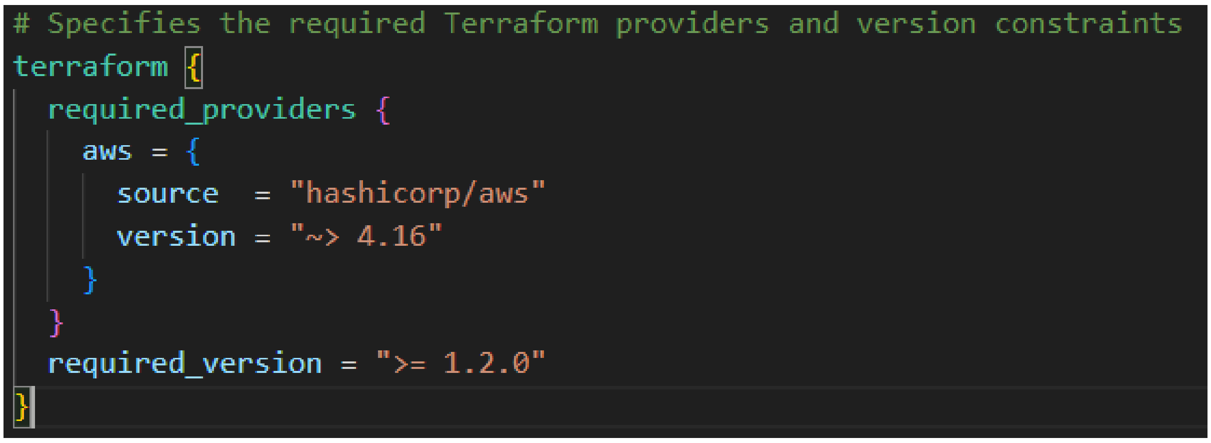


```
PS C:\Users\loren> aws configure
AWS Access Key ID [None]: [default]
AWS Secret Access Key [None]: aws_access_key_id=ASIA2XRTA2LDL7TSRBQI
Default region name [None]: aws_secret_access_key=K53gvv9JnI0Yp22diAy0xsP5YllE3oZ4oS+3sI9h
Def[redacted]CIBEwfX9NjOWHtzz/jwie
EQh                                                                 iDDiKpFZh6hpIeluh2iqR
AnB                                                                 1y/TEYnOPvkcXuJG3IlvA
syD                                                                 trgKhC3OuxW/5CfI8yacp
aKk                                                                 I0Rrbx28ROJWa22A1yqi6
QNO                                                                 u/uz7t25QT9NaiTowdvGb
jmmryASiXbsbY73Rw30vlVGfsEPvyWciIMqB+IOI31DY+UZcw3EFPh3w7OvEb+j6s6/oDjasw+AgoZuHnA1aPD6i66Ft5NR485DVPG6NIcg=
PS C:\Users\loren> |
```

Creating our IaC

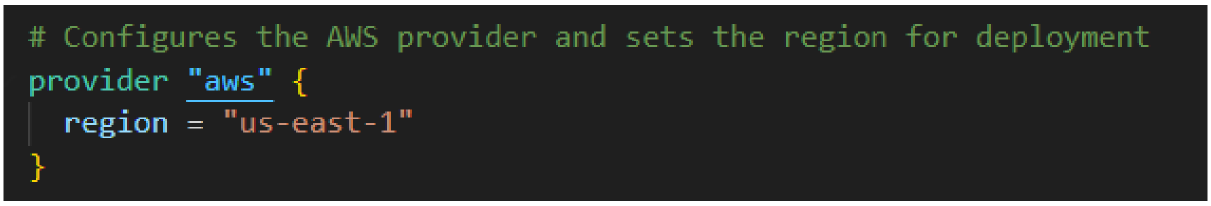
We created a new file called *main.tf* and inserted the following sections into our file.

1. *Configuring terraform to use AWS as provider*



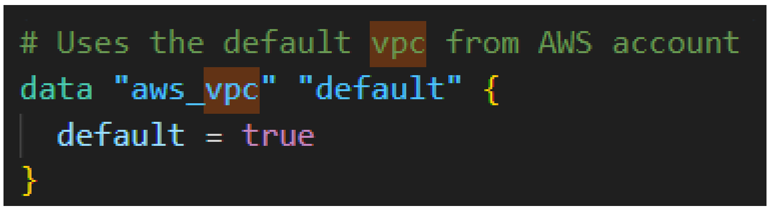
```
# Specifies the required Terraform providers and version constraints
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.16"
    }
  }
  required_version = ">= 1.2.0"
}
```

2. *Configure region*



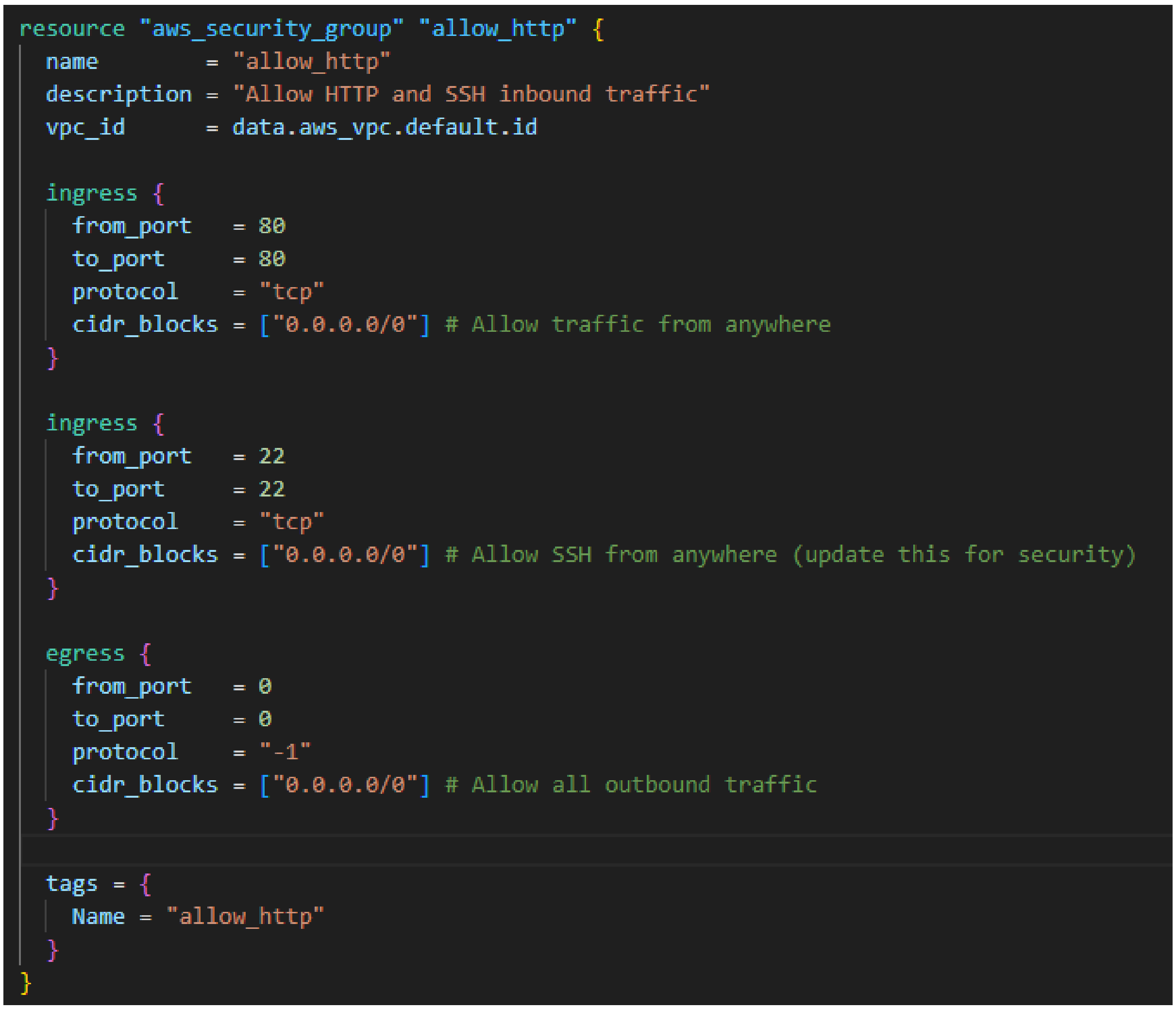
```
# Configures the AWS provider and sets the region for deployment
provider "aws" {
  region = "us-east-1"
}
```

3. *Use default VPC*



```
# Uses the default vpc from AWS account
data "aws_vpc" "default" {
  default = true
}
```

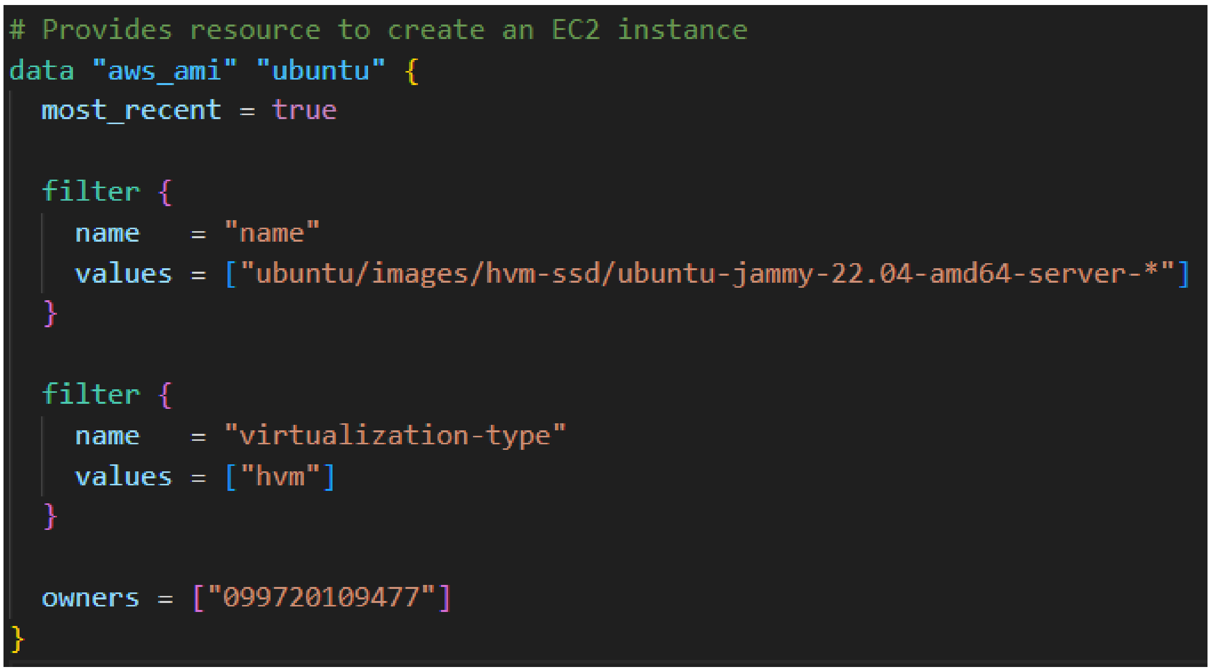
4. *Create a security group for HTTP and SSH ingress / open egress*



```
resource "aws_security_group" "allow_http" {
  name        = "allow_http"
  description = "Allow HTTP and SSH inbound traffic"
  vpc_id      = data.aws_vpc.default.id

  ingress {
    from_port   = 80
    to_port     = 80
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"] # Allow traffic from anywhere
  }

  ingress {
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"] # Allow SSH from anywhere (update this for security)
  }

  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"] # Allow all outbound traffic
  }

  tags = {
    Name = "allow_http"
  }
}
```

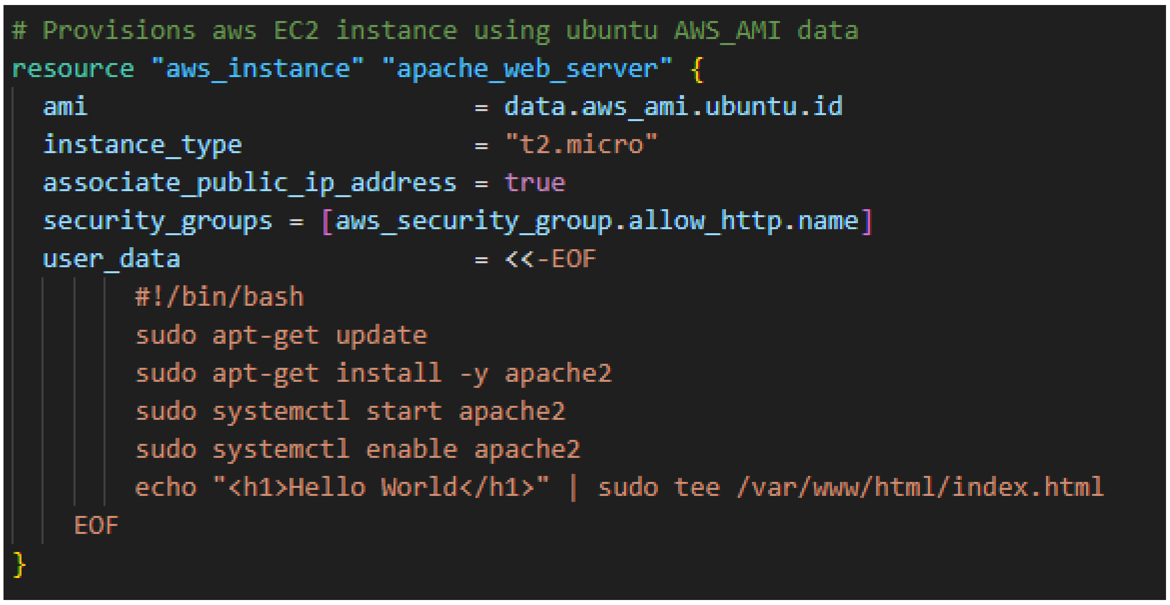
5. *Create an AWS EC2 Ubuntu instance*

Here, we fetch a *data* resource to use for our EC2 instance, which configures terraform to use the most recent Ubuntu Server AMI.



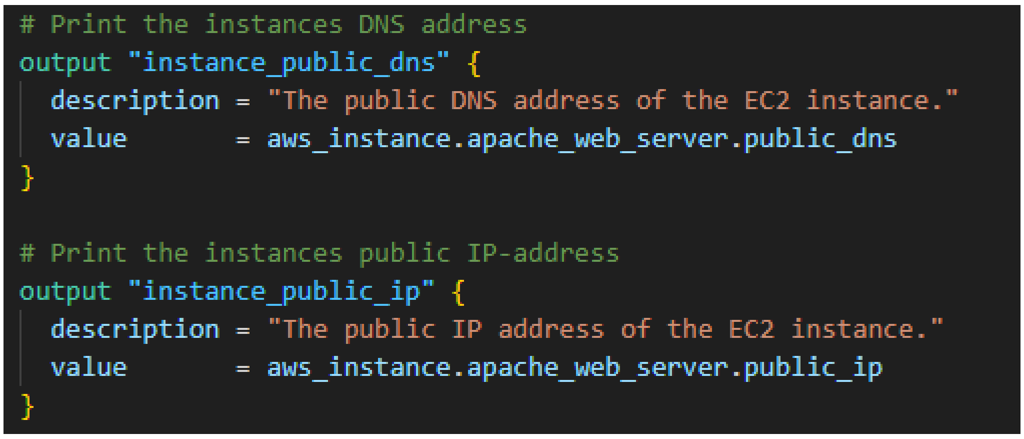
```
# Provides resource to create an EC2 instance
data "aws_ami" "ubuntu" {
  most_recent = true

  filter {
    name   = "name"
    values = ["ubuntu/images/hvm-ssd/ubuntu-jammy-22.04-amd64-server-*"]
  }

  filter {
    name   = "virtualization-type"
    values = ["hvm"]
  }

  owners = ["099720109477"]
}
```

Next, we configured the EC2 Instance with our created *data* resource and required specifications for *instance_type* and *associate_public_ip_address*.



```
# Provisions aws EC2 instance using ubuntu AWS_AMI data
resource "aws_instance" "apache_web_server" {
  ami                         = data.aws_ami.ubuntu.id
  instance_type               = "t2.micro"
  associate_public_ip_address = true
  security_groups = [aws_security_group.allow_http.name]
  user_data                   = <<-EOF
      #!/bin/bash
      sudo apt-get update
      sudo apt-get install -y apache2
      sudo systemctl start apache2
      sudo systemctl enable apache2
      echo "<h1>Hello World</h1>" | sudo tee /var/www/html/index.html
    EOF
}
```

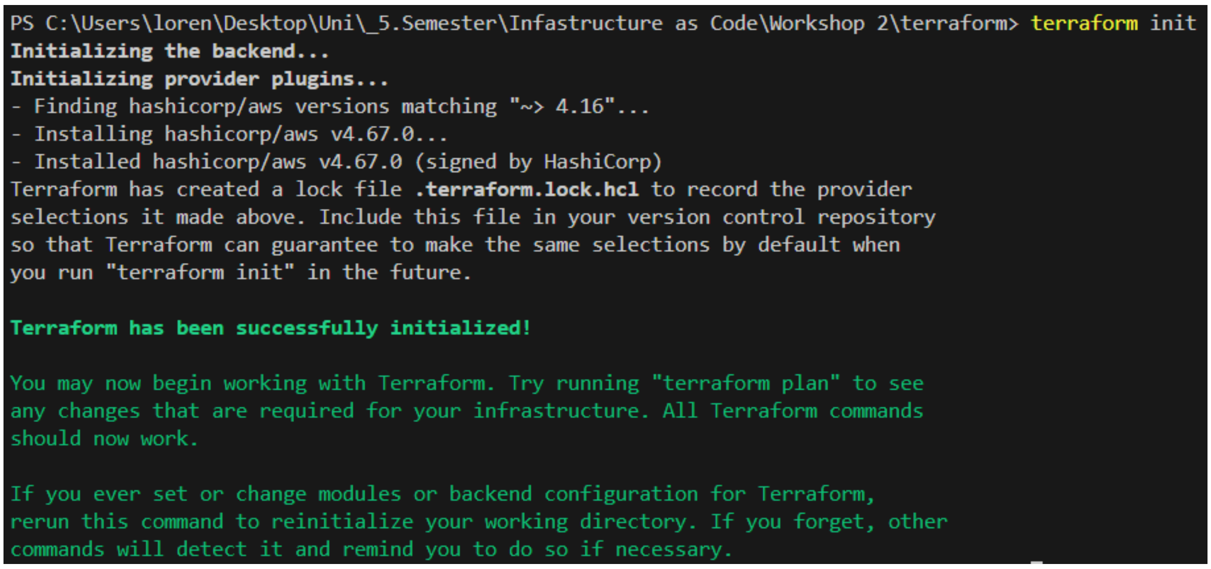
Lastly, we made sure our instance's DNS address and public IP address are printed after completion.



```
# Print the instances DNS address
output "instance_public_dns" {
  description = "The public DNS address of the EC2 instance."
  value       = aws_instance.apache_web_server.public_dns
}

# Print the instances public IP-address
output "instance_public_ip" {
  description = "The public IP address of the EC2 instance."
  value       = aws_instance.apache_web_server.public_ip
}
```

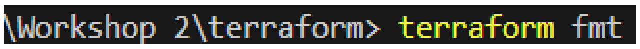
Running our IaC setup

We ran terraform init to download the AWS provider and set up the Terraform backend.



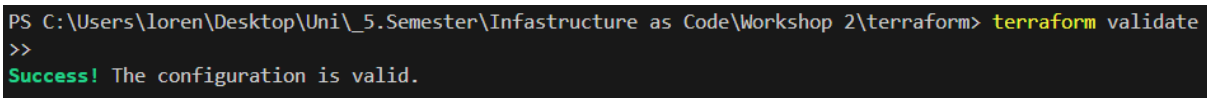
```
PS C:\Users\loren\Desktop\Uni\_5.Semester\Infastructure as Code\Workshop 2\terraform> terraform init
Initializing the backend...
Initializing provider plugins...
- Finding hashicorp/aws versions matching "~> 4.16"...
- Installing hashicorp/aws v4.67.0...
- Installed hashicorp/aws v4.67.0 (signed by HashiCorp)
Terraform has created a lock file .terraform.lock.hcl to record the provider
selections it made above. Include this file in your version control repository
so that Terraform can guarantee to make the same selections by default when
you run "terraform init" in the future.

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.
```

Next, we formatted and validated our configuration.



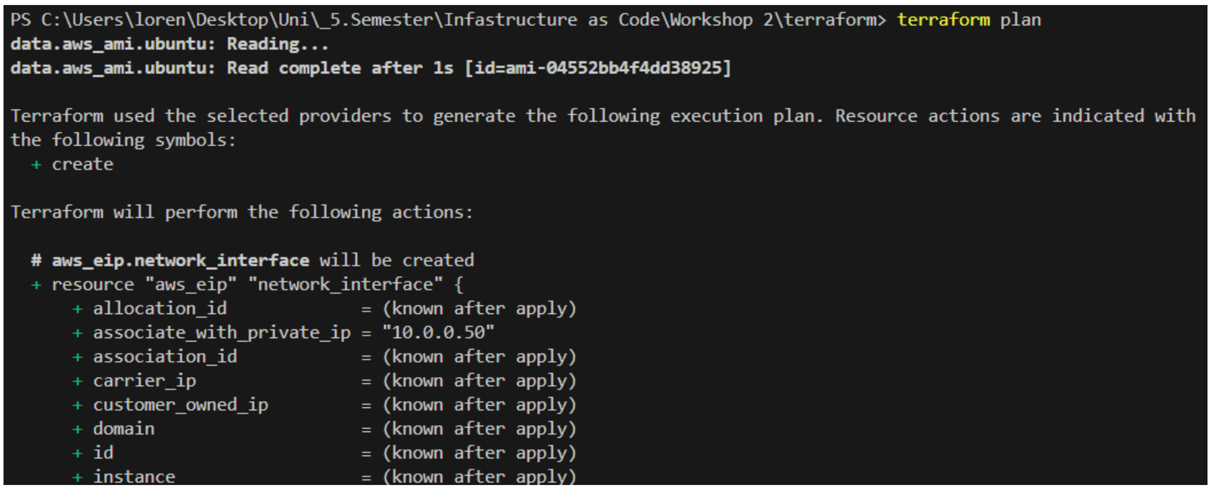
```
\Workshop 2\terraform> terraform fmt
```

After fixing some initial errors and misconfigurations in our resources, our validation passed.

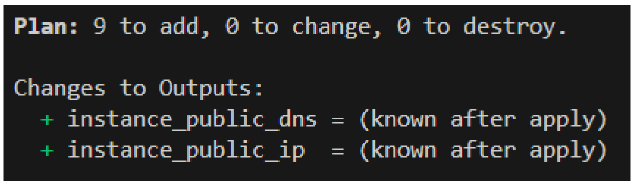


```
PS C:\Users\loren\Desktop\Uni\_5.Semester\Infastructure as Code\Workshop 2\terraform> terraform validate
>>
Success! The configuration is valid.
```

We then created an execution plan, which returned a looong list of changes to be applied to our AWS account.

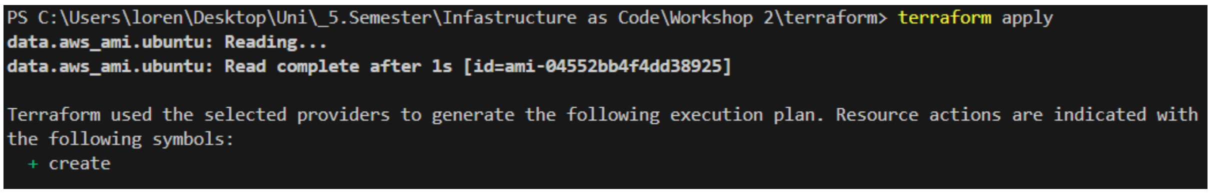


```
PS C:\Users\loren\Desktop\Uni\_5.Semester\Infastructure as Code\Workshop 2\terraform> terraform plan
data.aws_ami.ubuntu: Reading...
data.aws_ami.ubuntu: Read complete after 1s [id=ami-04552bb4f4dd38925]

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with
the following symbols:
  + create

Terraform will perform the following actions:

  # aws_eip.network_interface will be created
  + resource "aws_eip" "network_interface" {
      + allocation_id             = (known after apply)
      + associate_with_private_ip = "10.0.0.50"
      + association_id            = (known after apply)
      + carrier_ip                = (known after apply)
      + customer_owned_ip         = (known after apply)
      + domain                    = (known after apply)
      + id                        = (known after apply)
      + instance                  = (known after apply)
```



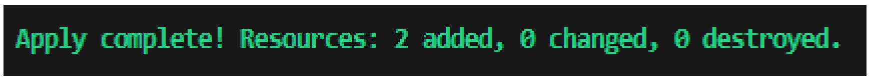
```
Plan: 9 to add, 0 to change, 0 to destroy.

Changes to Outputs:
  + instance_public_dns = (known after apply)
  + instance_public_ip  = (known after apply)
```

Lastly, we applied the configuration, which we had to confirm with a clear ‘yes’ via the terminal.



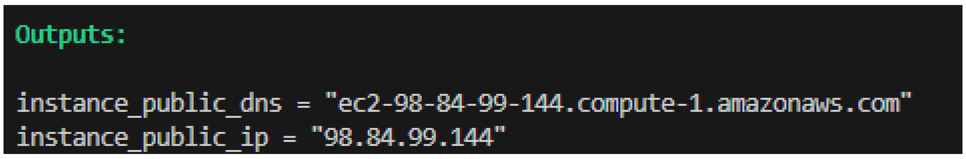
```
PS C:\Users\loren\Desktop\Uni\_5.Semester\Infastructure as Code\Workshop 2\terraform> terraform apply
data.aws_ami.ubuntu: Reading...
data.aws_ami.ubuntu: Read complete after 1s [id=ami-04552bb4f4dd38925]

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with
the following symbols:
  + create
```

We ran into some issues regarding our configuration, which had to be resolved. But after a few attempts, we got our infrastructure up and running.



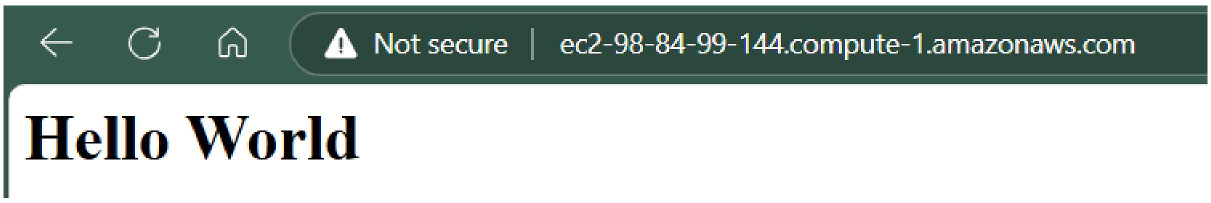
```
Apply complete! Resources: 2 added, 0 changed, 0 destroyed.
```

As confirmation, we received the DNS address as well as our instances public ip returned.



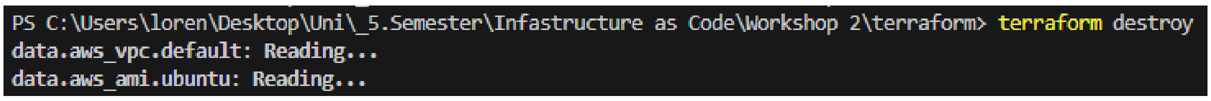
```
Outputs:

instance_public_dns = "ec2-98-84-99-144.compute-1.amazonaws.com"
instance_public_ip = "98.84.99.144"
```

Aaaaaand, there we had our hello world website, which we could access using the public DNS address.

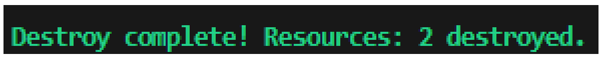


Hello World

Lastly, we destroyed our instance.



```
PS C:\Users\loren\Desktop\Uni\_5.Semester\Infastructure as Code\Workshop 2\terraform> terraform destroy
data.aws_vpc.default: Reading...
data.aws_ami.ubuntu: Reading...
```



```
Destroy complete! Resources: 2 destroyed.
```

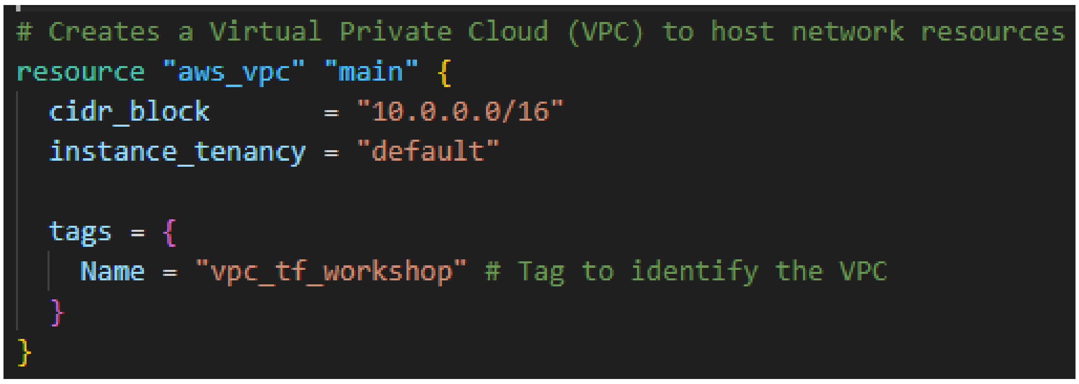
Part 2: Advanced Terraform Configuration

To create a more advanced configuration, we started off with what we created in the first part of our workshop.

Changing our configuration file

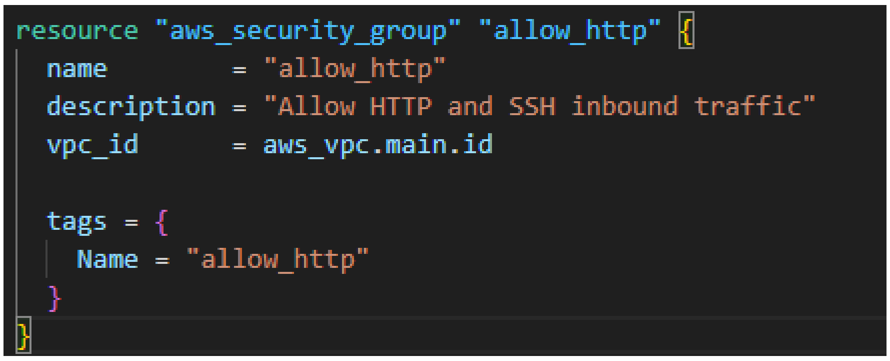
We made the following additions and changes to our main.tf.

1. *Create our own VPC, instead of using the default VPC*

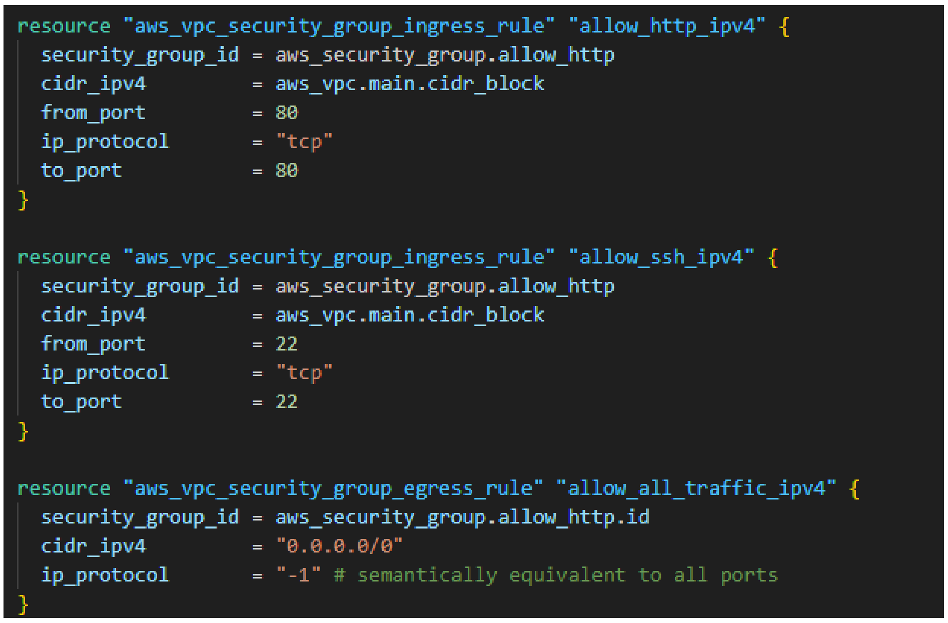


```
# Creates a Virtual Private Cloud (VPC) to host network resources
resource "aws_vpc" "main" {
  cidr_block       = "10.0.0.0/16"
  instance_tenancy = "default"

  tags = {
    Name = "vpc_tf_workshop" # Tag to identify the VPC
  }
}
```

2. *Revise our security groups to use ingress and egress resources*

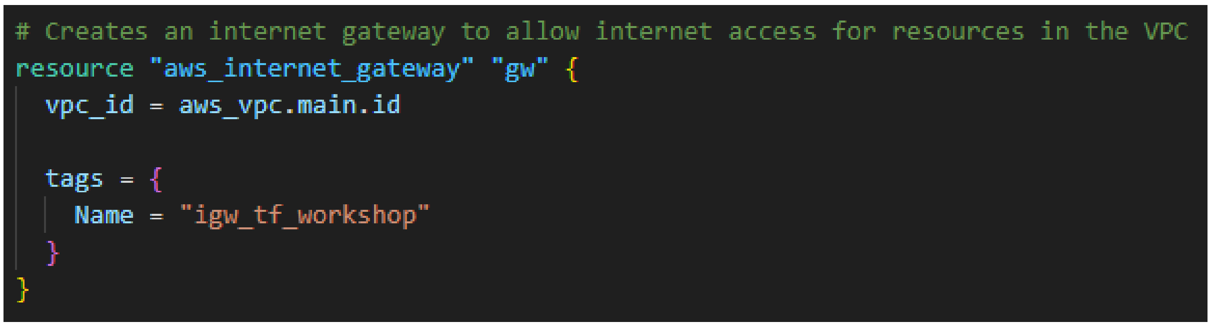


```
resource "aws_security_group" "allow_http" {
  name        = "allow_http"
  description = "Allow HTTP and SSH inbound traffic"
  vpc_id      = aws_vpc.main.id

  tags = {
    Name = "allow_http"
  }
}
```



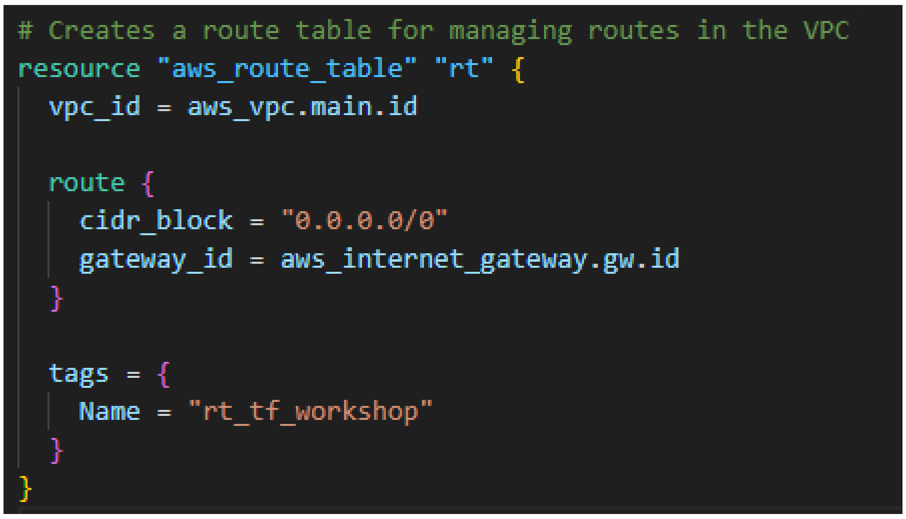
```
resource "aws_vpc_security_group_ingress_rule" "allow_http_ipv4" {
  security_group_id = aws_security_group.allow_http
  cidr_ipv4         = aws_vpc.main.cidr_block
  from_port         = 80
  ip_protocol       = "tcp"
  to_port           = 80
}

resource "aws_vpc_security_group_ingress_rule" "allow_ssh_ipv4" {
  security_group_id = aws_security_group.allow_http
  cidr_ipv4         = aws_vpc.main.cidr_block
  from_port         = 22
  ip_protocol       = "tcp"
  to_port           = 22
}

resource "aws_vpc_security_group_egress_rule" "allow_all_traffic_ipv4" {
  security_group_id = aws_security_group.allow_http.id
  cidr_ipv4         = "0.0.0.0/0"
  ip_protocol       = "-1" # semantically equivalent to all ports
}
```

3. *Create an internet gateway for our VPC*



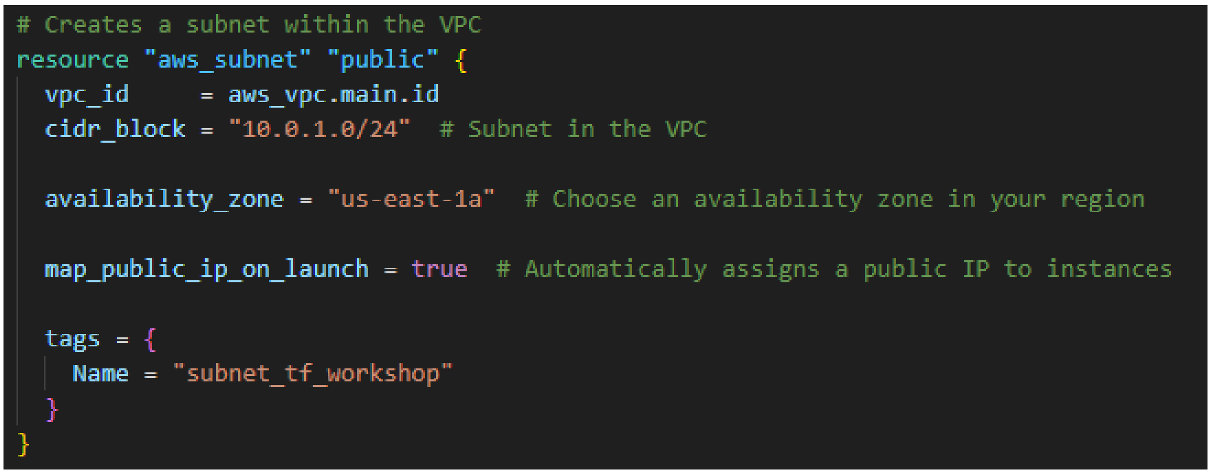
```
# Creates an internet gateway to allow internet access for resources in the VPC
resource "aws_internet_gateway" "gw" {
  vpc_id = aws_vpc.main.id

  tags = {
    Name = "igw_tf_workshop"
  }
}
```

4. *Create a route table for our VPC*



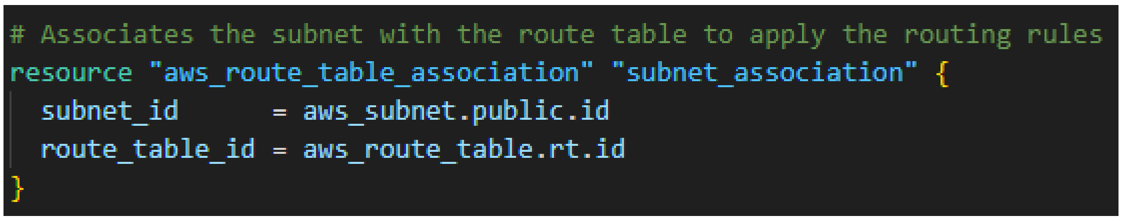
```
# Creates a route table for managing routes in the VPC
resource "aws_route_table" "rt" {
  vpc_id = aws_vpc.main.id

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

  tags = {
    Name = "rt_tf_workshop"
  }
}
```

5. *Create a public subnet in our VPC*



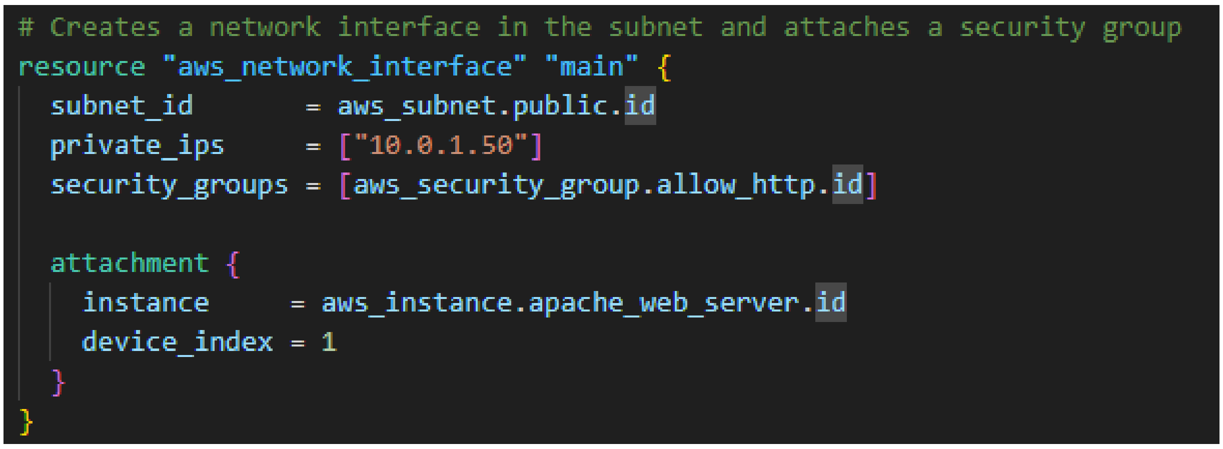
```
# Creates a subnet within the VPC
resource "aws_subnet" "public" {
  vpc_id     = aws_vpc.main.id
  cidr_block = "10.0.1.0/24"  # Subnet in the VPC

  availability_zone = "us-east-1a"  # Choose an availability zone in your region

  map_public_ip_on_launch = true  # Automatically assigns a public IP to instances

  tags = {
    Name = "subnet_tf_workshop"
  }
}
```

6. *Associate the subnet with the route table*



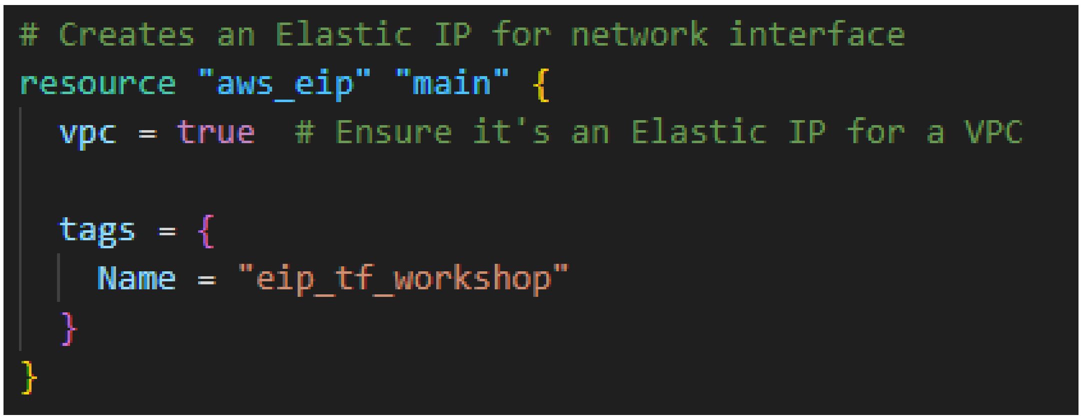
```
# Associates the subnet with the route table to apply the routing rules
resource "aws_route_table_association" "subnet_association" {
  subnet_id      = aws_subnet.public.id
  route_table_id = aws_route_table.rt.id
}
```

7. *Create a network interface in our public subnet*



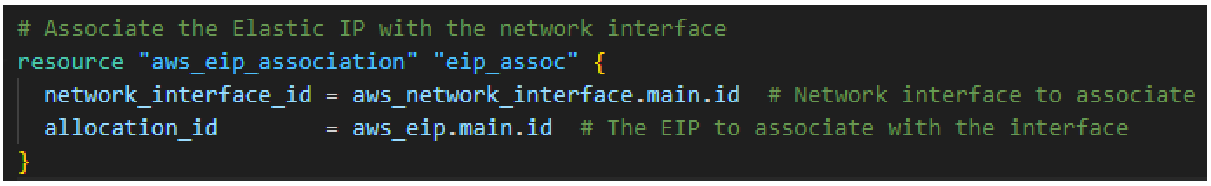
```
# Creates a network interface in the subnet and attaches a security group
resource "aws_network_interface" "main" {
  subnet_id       = aws_subnet.public.id
  private_ips     = ["10.0.1.50"]
  security_groups = [aws_security_group.allow_http.id]

  attachment {
    instance     = aws_instance.apache_web_server.id
    device_index = 1
  }
}
```

8. *Create an elastic IP for our network interface*



```
# Creates an Elastic IP for network interface
resource "aws_eip" "main" {
  vpc = true  # Ensure it's an Elastic IP for a VPC

  tags = {
    Name = "eip_tf_workshop"
  }
}
```

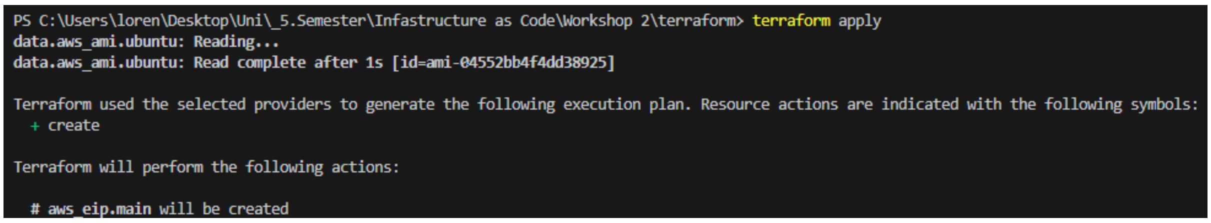
9. *Associate EIP with network interface*



```
# Associate the Elastic IP with the network interface
resource "aws_eip_association" "eip_assoc" {
  network_interface_id = aws_network_interface.main.id  # Network interface to associate
  allocation_id        = aws_eip.main.id  # The EIP to associate with the interface
}
```

Running our IaC

After adding all those configurations, and adapting variables to use our newly created instances, instead of old owns (e.g. default vpc), we were reading to apply.

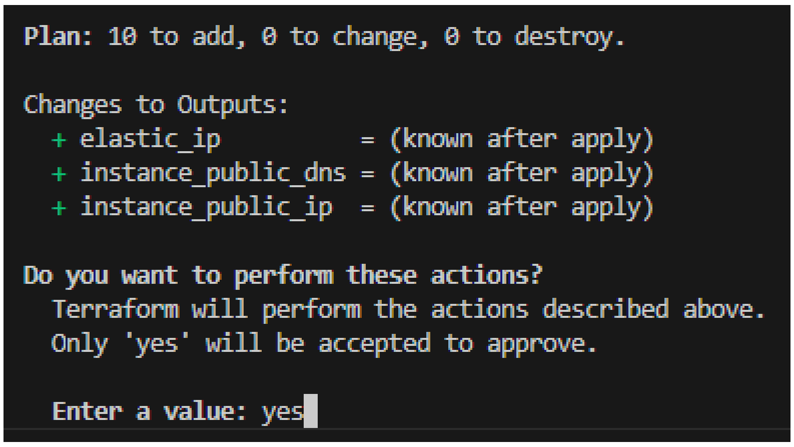


```
PS C:\Users\loren\Desktop\Uni\_5.Semester\Infrastructure as Code\Workshop 2\terraform> terraform apply
data.aws_ami.ubuntu: Reading...
data.aws_ami.ubuntu: Read complete after 1s [id=ami-04552bb4f4dd38925]

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # aws_eip.main will be created
```

.

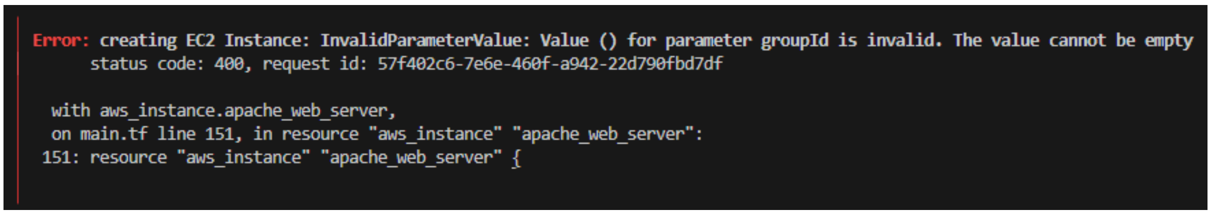
.

.



```
Plan: 10 to add, 0 to change, 0 to destroy.

Changes to Outputs:
  + elastic_ip          = (known after apply)
  + instance_public_dns = (known after apply)
  + instance_public_ip  = (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
```

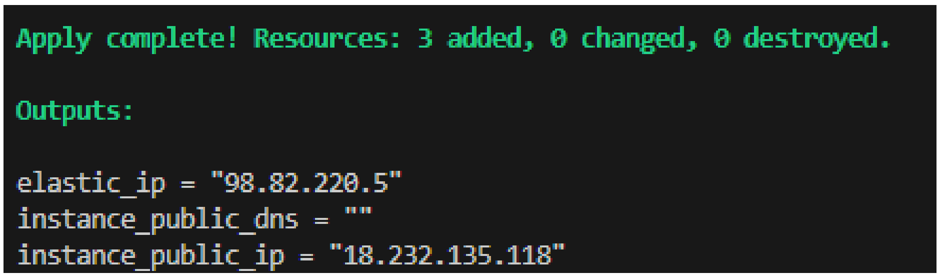
Again, we received an error here and there.



```
Error: creating EC2 Instance: InvalidParameterValue: Value () for parameter groupId is invalid. The value cannot be empty
       status code: 400, request id: 57f402c6-7e6e-460f-a942-22d790fbd7df

  with aws_instance.apache_web_server,
  on main.tf line 151, in resource "aws_instance" "apache_web_server":
 151: resource "aws_instance" "apache_web_server" {
```

Since we now used our own VPC, we had to fix the security group association of our Apache server instance from *security_groups* to *vpc_security_group_ids*. Also, we had to add our Apache server instance to our public subnet, which we initially forgot.

And then, it worked.



```
Apply complete! Resources: 3 added, 0 changed, 0 destroyed.

Outputs:

elastic_ip = "98.82.220.5"
instance_public_dns = ""
instance_public_ip = "18.232.135.118"
```

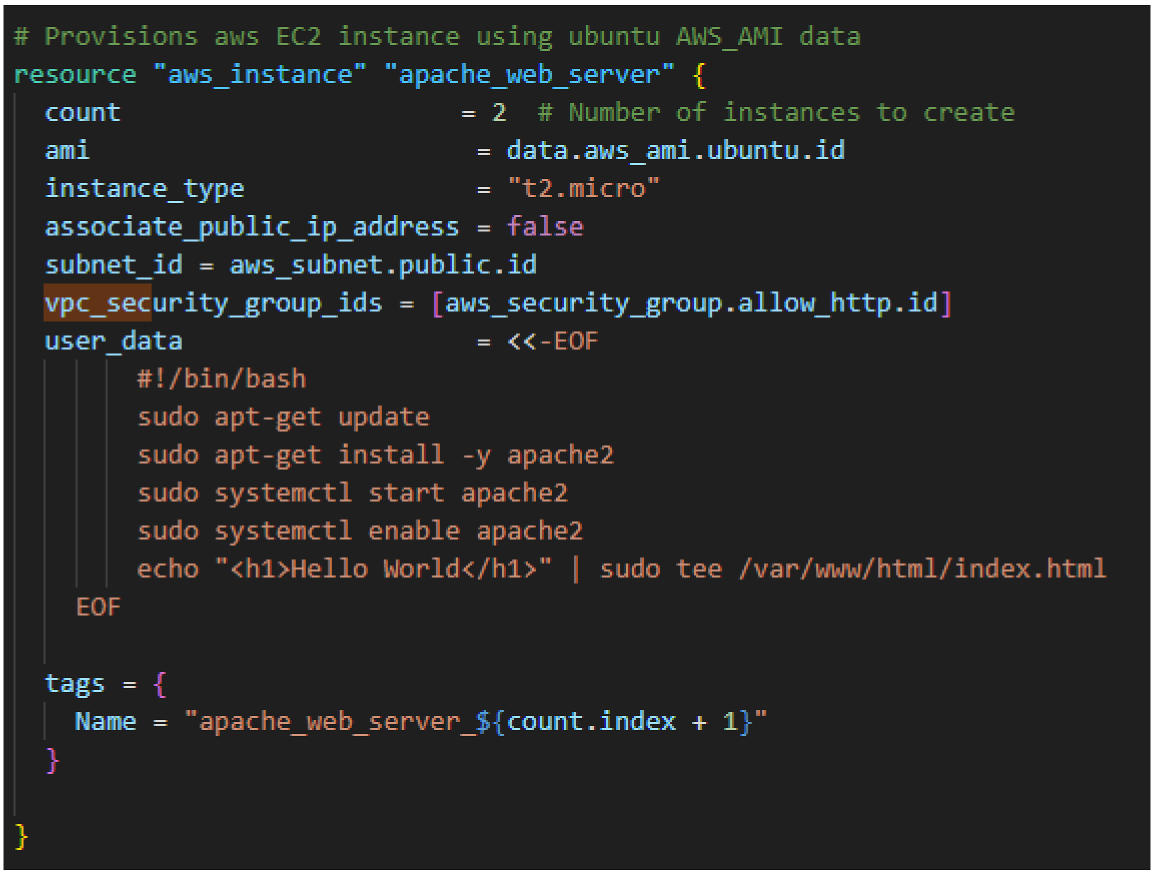
Part 3: Terraform with CI/CD

First, we had to modify our setup.

Adding Configurations to our main.tf

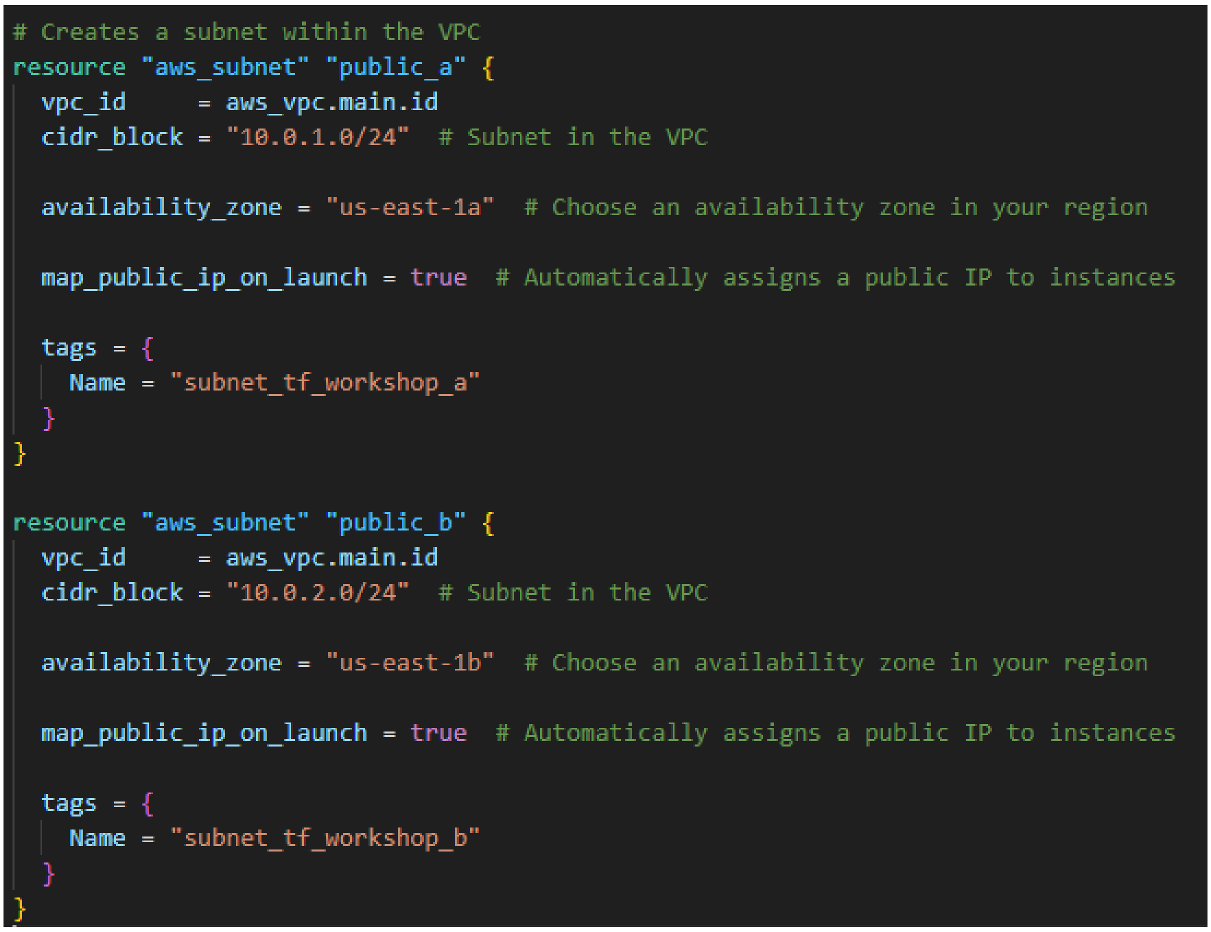
1. *Create multiple instances*

This we achieved by adding a count of 2 to our Apache server instance. Also, we added name tags to give our two instances different names.



```
# Provisions aws EC2 instance using ubuntu AWS_AMI data
resource "aws_instance" "apache_web_server" {
  count                      = 2  # Number of instances to create
  ami                         = data.aws_ami.ubuntu.id
  instance_type               = "t2.micro"
  associate_public_ip_address = false
  subnet_id = aws_subnet.public.id
  vpc_security_group_ids = [aws_security_group.allow_http.id]
  user_data                   = <<-EOF
        #!/bin/bash
        sudo apt-get update
        sudo apt-get install -y apache2
        sudo systemctl start apache2
        sudo systemctl enable apache2
        echo "<h1>Hello World</h1>" | sudo tee /var/www/html/index.html
    EOF

  tags = {
    Name = "apache_web_server_${count.index + 1}"
  }

}
```

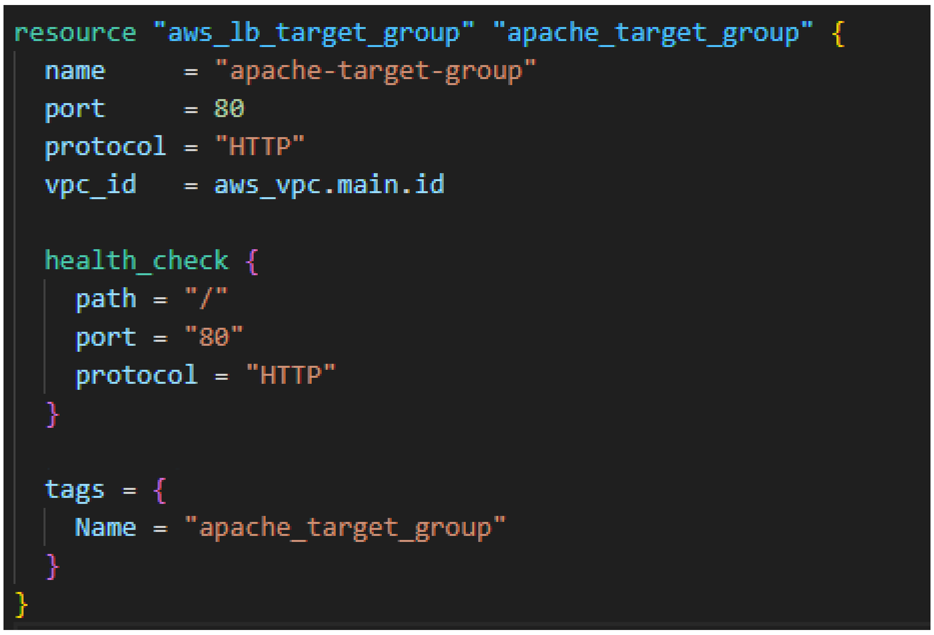
2. Create Multiple Subnets for Load Balancing



```
# Creates a subnet within the VPC
resource "aws_subnet" "public_a" {
  vpc_id     = aws_vpc.main.id
  cidr_block = "10.0.1.0/24"  # Subnet in the VPC

  availability_zone = "us-east-1a"  # Choose an availability zone in your region

  map_public_ip_on_launch = true  # Automatically assigns a public IP to instances

  tags = {
    Name = "subnet_tf_workshop_a"
  }
}

resource "aws_subnet" "public_b" {
  vpc_id     = aws_vpc.main.id
  cidr_block = "10.0.2.0/24"  # Subnet in the VPC

  availability_zone = "us-east-1b"  # Choose an availability zone in your region

  map_public_ip_on_launch = true  # Automatically assigns a public IP to instances

  tags = {
    Name = "subnet_tf_workshop_b"
  }
}
```

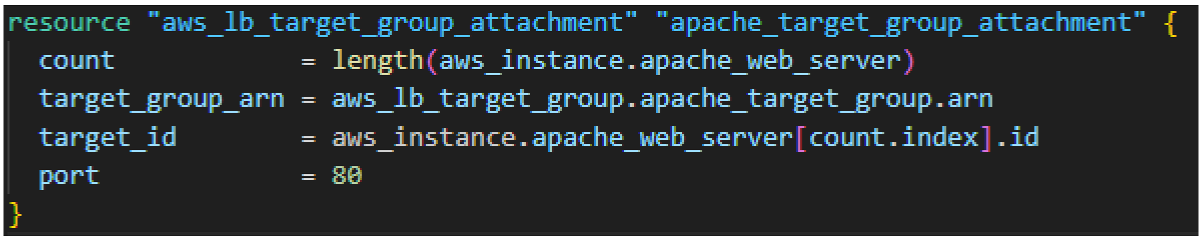
3. Configure Load Balancer Target Groups

We created a target group for our load balancer, using port 80



```
resource "aws_lb_target_group" "apache_target_group" {
  name     = "apache-target-group"
  port     = 80
  protocol = "HTTP"
  vpc_id   = aws_vpc.main.id

  health_check {
    path = "/"
    port = "80"
    protocol = "HTTP"
  }

  tags = {
    Name = "apache_target_group"
  }
}
```

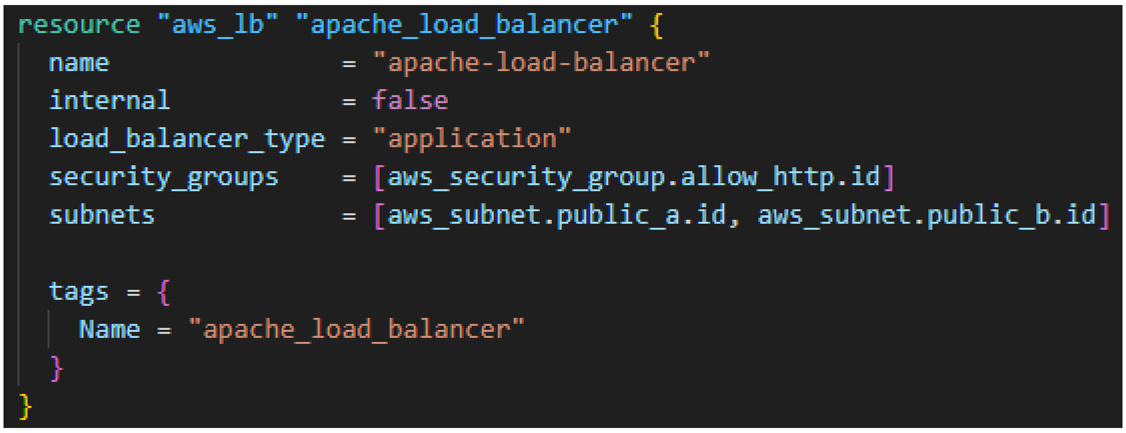
Then, we added the configuration to attach all of our Apache server instances to our target group.



```
resource "aws_lb_target_group_attachment" "apache_target_group_attachment" {
  count            = length(aws_instance.apache_web_server)
  target_group_arn = aws_lb_target_group.apache_target_group.arn
  target_id        = aws_instance.apache_web_server[count.index].id
  port             = 80
}
```

4. Configure Load Balancer

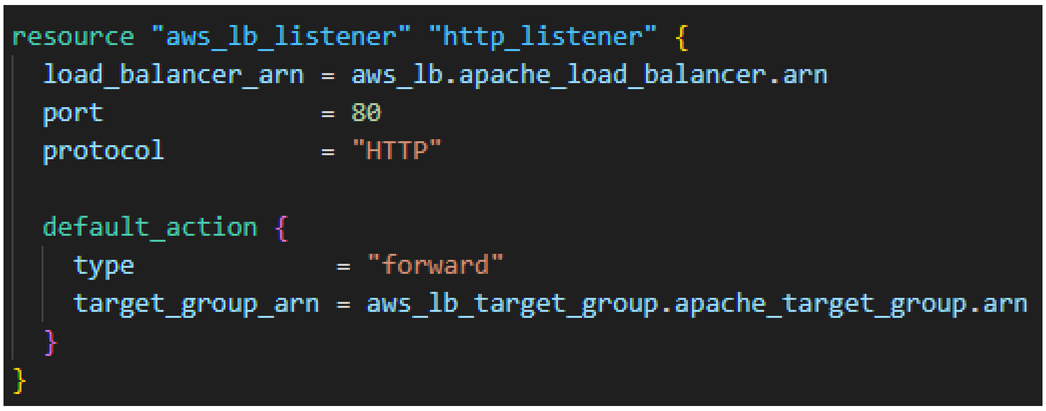
Then, we created the load balancer resource itself.



```
resource "aws_lb" "apache_load_balancer" {
  name               = "apache-load-balancer"
  internal           = false
  load_balancer_type = "application"
  security_groups    = [aws_security_group.allow_http.id]
  subnets            = [aws_subnet.public_a.id, aws_subnet.public_b.id]

  tags = {
    Name = "apache_load_balancer"
  }
}
```

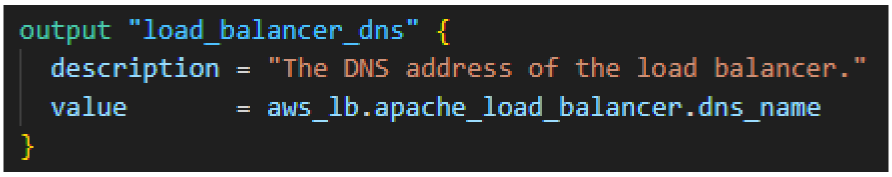
5. Add a Listener to Load Balancer

Lastly, we added a listener for our load balancer, to define how incoming requests are handled.



```
resource "aws_lb_listener" "http_listener" {
  load_balancer_arn = aws_lb.apache_load_balancer.arn
  port              = 80
  protocol          = "HTTP"

  default_action {
    type             = "forward"
    target_group_arn = aws_lb_target_group.apache_target_group.arn
  }
}
```

6. Output for Load Balancer DNS Record



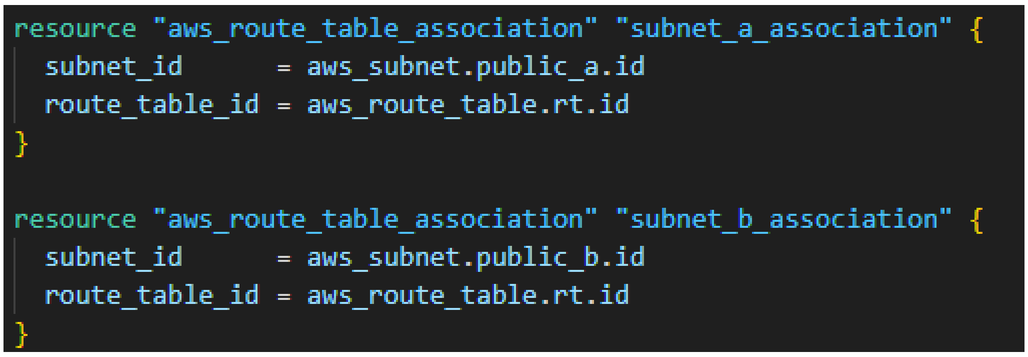
```
output "load_balancer_dns" {
  description = "The DNS address of the load balancer."
  value       = aws_lb.apache_load_balancer.dns_name
}
```

7. *Remove Network Interface and Elastic IP*

Since we are using a load balancer now, we had to remove our network interface and our elastic IP, so we just commented them out.

8. *Reconfigure Route Table for Additional Subnet*

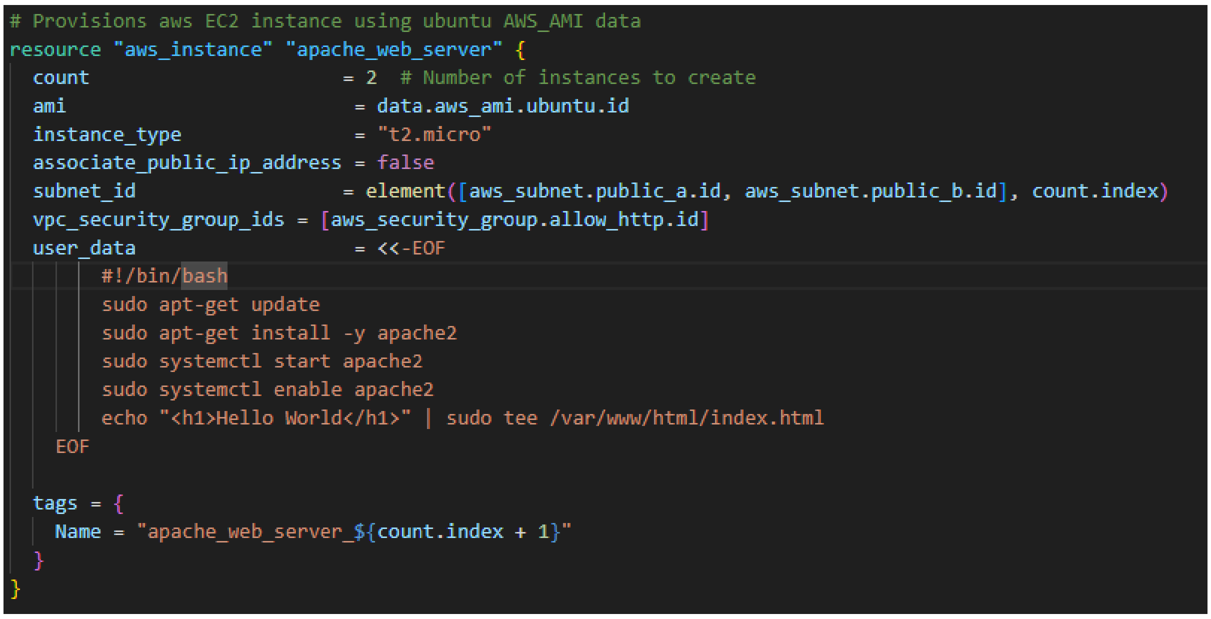
We had to associate our second subnet to our route table.



```
resource "aws_route_table_association" "subnet_a_association" {
  subnet_id      = aws_subnet.public_a.id
  route_table_id = aws_route_table.rt.id
}

resource "aws_route_table_association" "subnet_b_association" {
  subnet_id      = aws_subnet.public_b.id
  route_table_id = aws_route_table.rt.id
}
```

9. *Reconfigure our Instances for our Second AZ and Subnet*

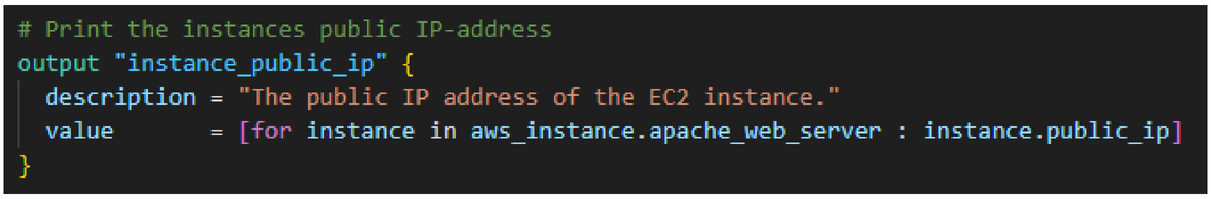
We had to make sure our instances were distributed across our two subnets using the element function of terraform.



```
# Provisions aws EC2 instance using ubuntu AWS_AMI data
resource "aws_instance" "apache_web_server" {
  count                      = 2  # Number of instances to create
  ami                         = data.aws_ami.ubuntu.id
  instance_type               = "t2.micro"
  associate_public_ip_address = false
  subnet_id                  = element([aws_subnet.public_a.id, aws_subnet.public_b.id], count.index)
  vpc_security_group_ids = [aws_security_group.allow_http.id]
  user_data                   = <<-EOF
        #!/bin/bash
        sudo apt-get update
        sudo apt-get install -y apache2
        sudo systemctl start apache2
        sudo systemctl enable apache2
        echo "<h1>Hello World</h1>" | sudo tee /var/www/html/index.html
    EOF

  tags = {
    Name = "apache_web_server_${count.index + 1}"
  }
}
```

10. *Print all instances IPs*

We also adjusted our outputs, to print all resources' IP addresses.

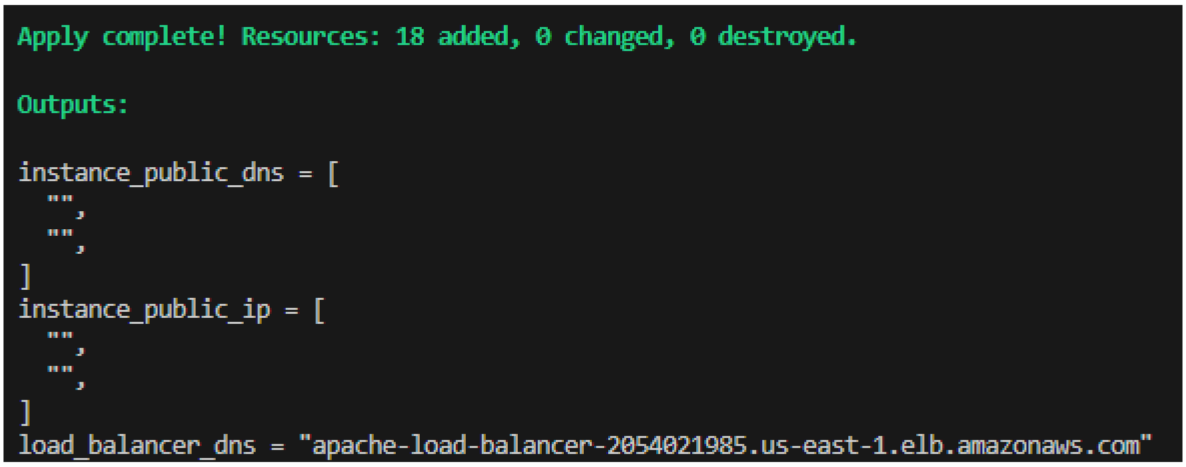


```
# Print the instances public IP-address
output "instance_public_ip" {
  description = "The public IP address of the EC2 instance."
  value       = [for instance in aws_instance.apache_web_server : instance.public_ip]
}
```

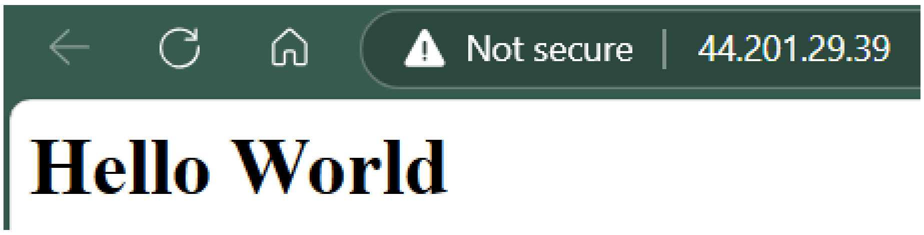
Running our IaC

We tried applying our changes, and once again ran into some issues. There suddenly were multiple instances, which meant we had to reconfigure how we attach and associate them.

After some troubleshooting however, we were able to get our instances up and running.



```
Apply complete! Resources: 18 added, 0 changed, 0 destroyed.

Outputs:

instance_public_dns = [
  "",
  "",
]
instance_public_ip = [
  "",
  "",
]
load_balancer_dns = "apache-load-balancer-2054021985.us-east-1.elb.amazonaws.com"
```

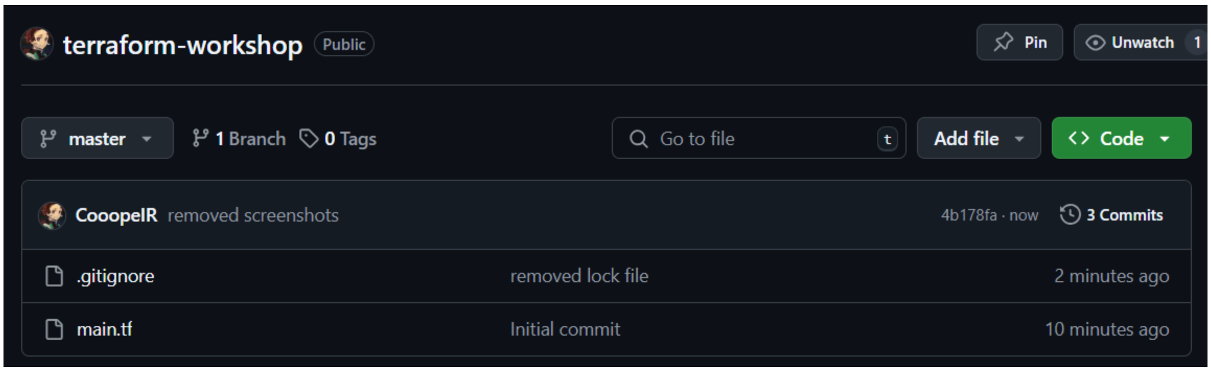


Not secure | 44.201.29.39

Hello World

Configuring GitHub Actions

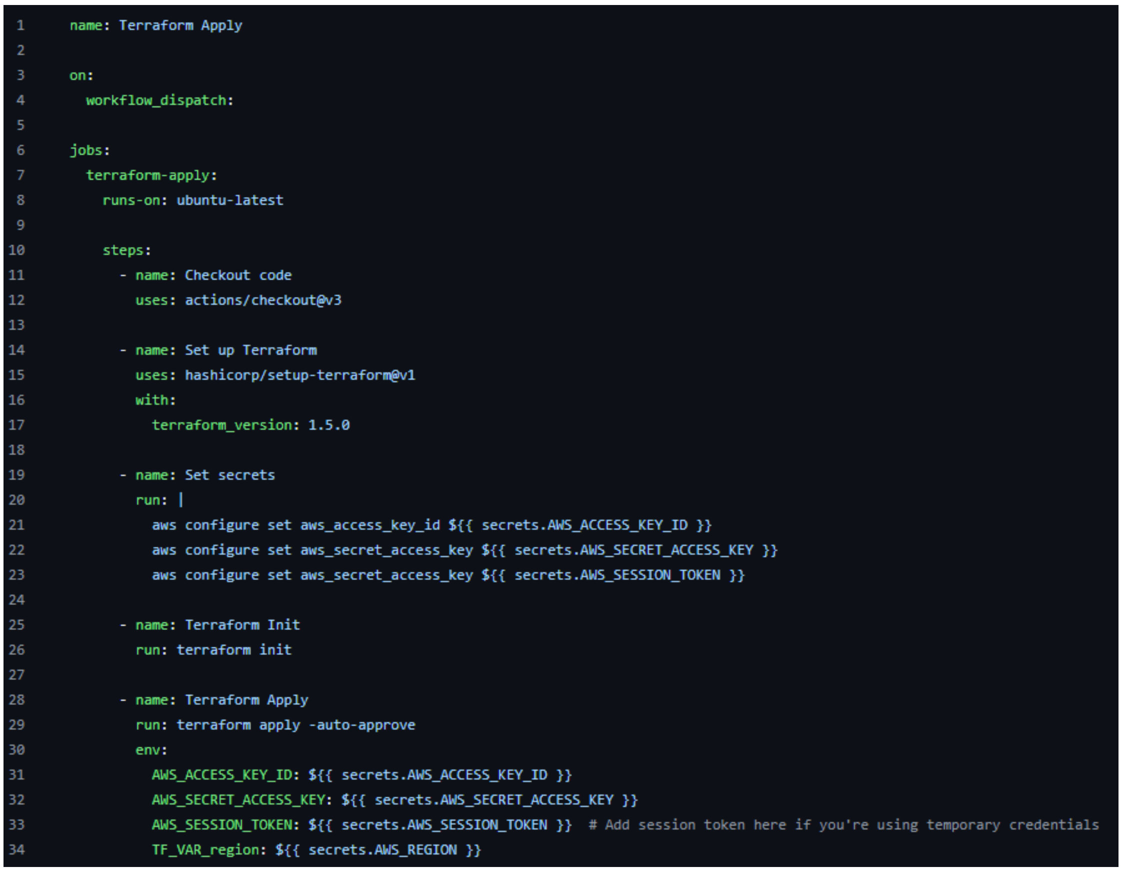
We created a repository for our terraform configuration.



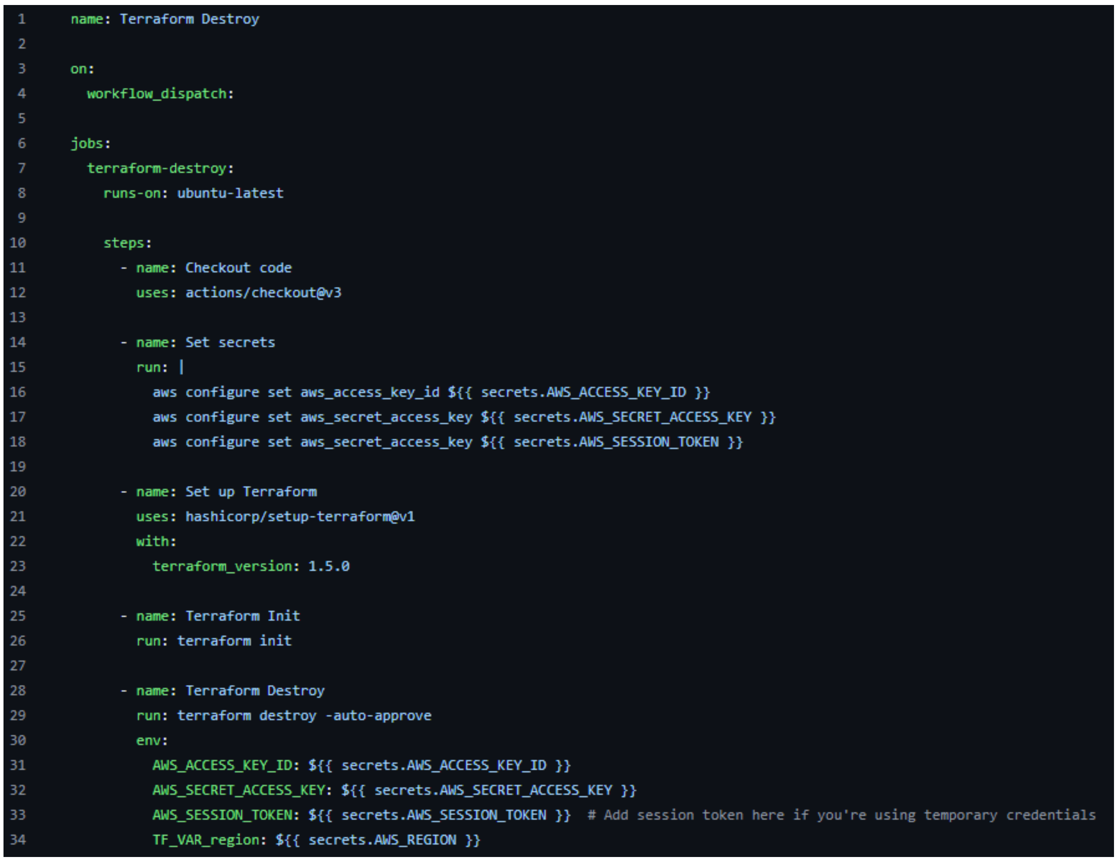
terraform-workshop Public

CooopelR removed screenshots		4b178fa · now 3 Commits
.gitignore	removed lock file	2 minutes ago
main.tf	Initial commit	10 minutes ago

We added two YAML files, one for the application of our infrastructure, and one for its destruction.

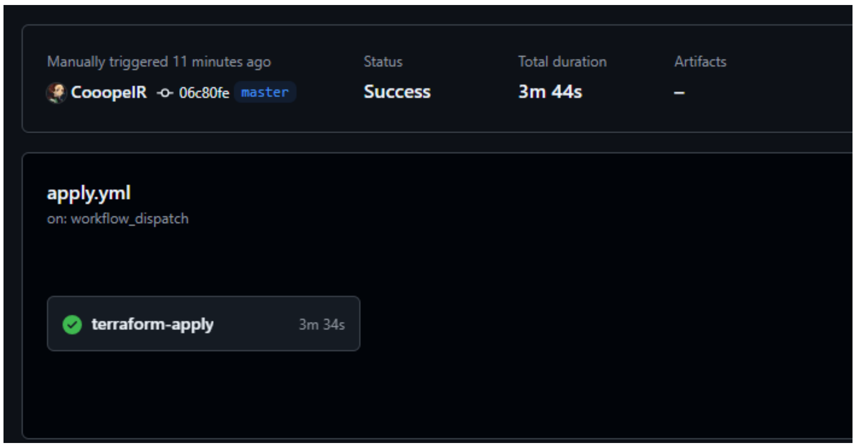


```yaml
name: Terraform Apply

on:
  workflow_dispatch:

jobs:
  terraform-apply:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Set up Terraform
        uses: hashicorp/setup-terraform@v1
        with:
          terraform_version: 1.5.0

      - name: Set secrets
        run: |
          aws configure set aws_access_key_id ${{ secrets.AWS_ACCESS_KEY_ID }}
          aws configure set aws_secret_access_key ${{ secrets.AWS_SECRET_ACCESS_KEY }}
          aws configure set aws_secret_access_key ${{ secrets.AWS_SESSION_TOKEN }}

      - name: Terraform Init
        run: terraform init

      - name: Terraform Apply
        run: terraform apply -auto-approve
        env:
          AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}
          AWS_SECRET_ACCESS_KEY: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
          AWS_SESSION_TOKEN: ${{ secrets.AWS_SESSION_TOKEN }}  # Add session token here if you're using temporary credentials
          TF_VAR_region: ${{ secrets.AWS_REGION }}
```



```yaml
name: Terraform Destroy

on:
  workflow_dispatch:

jobs:
  terraform-destroy:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Set secrets
        run: |
          aws configure set aws_access_key_id ${{ secrets.AWS_ACCESS_KEY_ID }}
          aws configure set aws_secret_access_key ${{ secrets.AWS_SECRET_ACCESS_KEY }}
          aws configure set aws_secret_access_key ${{ secrets.AWS_SESSION_TOKEN }}

      - name: Set up Terraform
        uses: hashicorp/setup-terraform@v1
        with:
          terraform_version: 1.5.0

      - name: Terraform Init
        run: terraform init

      - name: Terraform Destroy
        run: terraform destroy -auto-approve
        env:
          AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}
          AWS_SECRET_ACCESS_KEY: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
          AWS_SESSION_TOKEN: ${{ secrets.AWS_SESSION_TOKEN }}  # Add session token here if you're using temporary credentials
          TF_VAR_region: ${{ secrets.AWS_REGION }}
```

We then ran our workflows to make sure they worked – this involved a lot of debugging again, as our authentication with AWS did not work in the beginning.

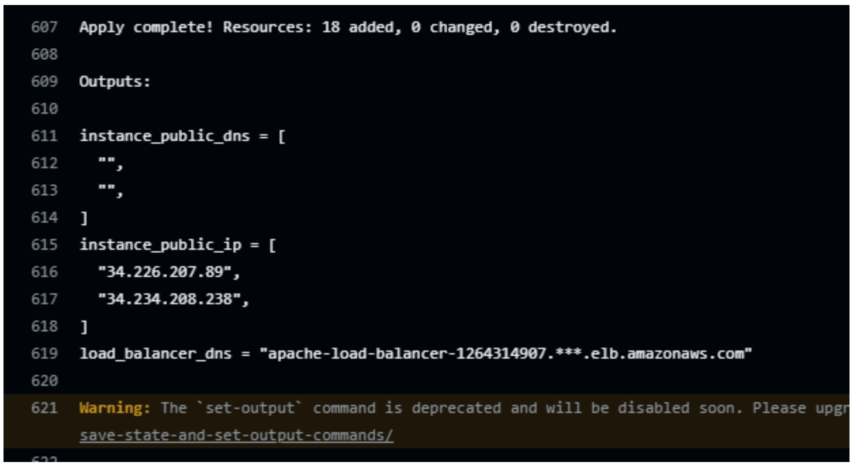


Manually triggered 11 minutes ago
CooopelR 06c80fe master

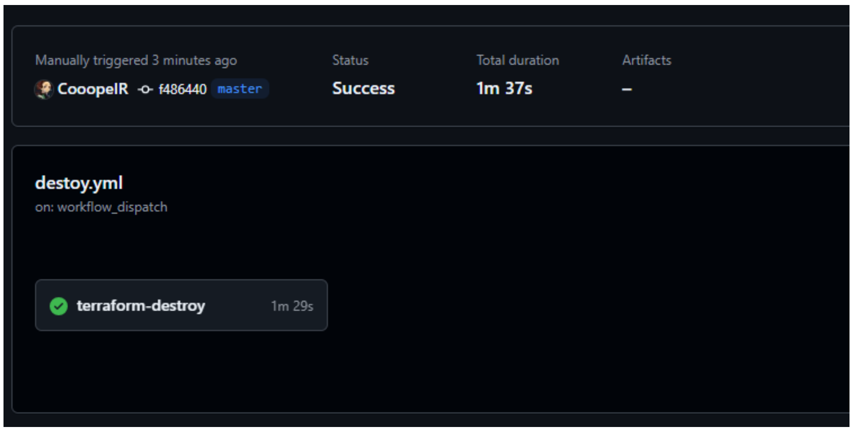
Status: Success
Total duration: 3m 44s
Artifacts: –

apply.yml
on: workflow_dispatch

terraform-apply 3m 34s



```
Apply complete! Resources: 18 added, 0 changed, 0 destroyed.

Outputs:

instance_public_dns = [
  "",
  "",
]
instance_public_ip = [
  "34.226.207.89",
  "34.234.208.238",
]
load_balancer_dns = "apache-load-balancer-1264314907.***.elb.amazonaws.com"

Warning: The `set-output` command is deprecated and will be disabled soon. Please upgr
save-state-and-set-output-commands/
```

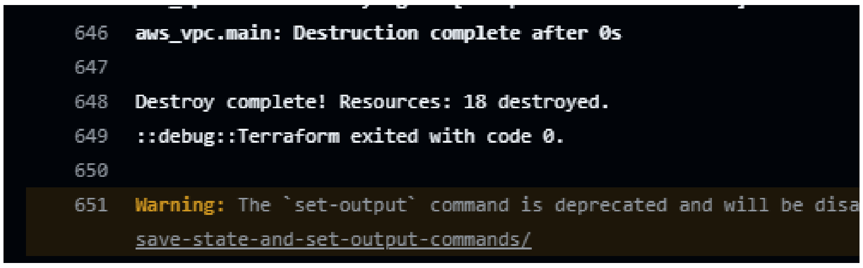


Manually triggered 3 minutes ago
CooopelR f486440 master

Status: Success
Total duration: 1m 37s
Artifacts: –

destoy.yml
on: workflow_dispatch

terraform-destroy 1m 29s



```
aws_vpc.main: Destruction complete after 0s

Destroy complete! Resources: 18 destroyed.
::debug::Terraform exited with code 0.

Warning: The `set-output` command is deprecated and will be disa
save-state-and-set-output-commands/
```

Question

Why do we need Terraform Cloud (or another backend) when we use CI/CD?

Answer: First, IaC Tools are great at quickly modifying the infrastructure used for our applications. We can simply add another resource and right away, our workflow will create the new instances for us, we don’t even have to trigger the creation (or destruction) ourselves using CI/CD pipelines.

Also, Terraform keeps track of our infrastructures state, which is helpful in cases where multiple developers or even teams work on the same system. Unintentional state changes will only be a minor problem that way. Having the state stored remotely with Terraform Cloud or in a git Repository allows us to keep the file a single source of truth and consistent all across.

Having our files accessible online also allows other applications to use our infrastructure configuration for building or testing purposes.

Lastly, it can improve security, as environment variables are encapsulated in the spot where they are needed.