# Load Balancing: Add an Application Load Balancer

**We'll cover the following**

- Objective
- Steps
- Adding the load balancer

## Objective

- Run our application on more than one EC2 instance.

## Steps

- Add an Application Load Balancer.

## Adding the load balancer

Until now, we have been relying on EC2's default network configuration, but by adding a load balancer we're going to have to explicitly define a VPC setup in our CloudFormation template.

We're also going to make sure that our two instances will be running in separate availability zones. This is a fundamental requirement for ensuring high availability when using EC2. We recommend James Hamilton's video Failures at Scale and How to Ignore Them from 2012 as a good introduction to how AWS thinks about availability zones and high availability.

The VPC and subnets will require IP address allocations, for which we need to use CIDR (Classless inter-domain routing) notation. A full discussion of CIDR concepts is beyond the scope of this book, but AWS has some documentation about this topic in VPC sizing and VPC design. For our purposes, we will be allocating our VPC addresses from one of the RFC 1918 private address ranges: `10.0.0.0 - 10.255.255.255`.

The CIDR notation for that whole address range is `10.0.0.0/8`. VPC allows us to allocate up to half of that range with a notation of `10.0.0.0/16`, which corresponds

to 65,536 unique IP addresses. We'll split it evenly into 4 subnets of 16,382 addresses each: `10.0.0.0/18`, `10.0.64.0/18`, `10.0.128.0/18`, and `10.0.192.0/18`. We're going to use two blocks of addresses right now, and the other two when we get to the topic of network security.

The following might look very complicated, but it's mostly boilerplate configuration. You will rarely need to change any of these settings.

So, let's start by adding our VPC and two subnets to our CloudFormation template.

```
VPC:
  Type: AWS::EC2::VPC
  Properties:
    CidrBlock: 10.0.0.0/16
    EnableDnsSupport: true
    EnableDnsHostnames: true
    Tags:
      - Key: Name
        Value: !Ref AWS::StackName

SubnetAZ1:
  Type: AWS::EC2::Subnet
  Properties:
    VpcId: !Ref VPC
    AvailabilityZone: !Select [ 0, !GetAZs '' ]
    CidrBlock: 10.0.0.0/18
    MapPublicIpOnLaunch: true <3>
    Tags:
      - Key: Name
        Value: !Ref AWS::StackName
      - Key: AZ
        Value: !Select [ 0, !GetAZs '' ]

SubnetAZ2:
  Type: AWS::EC2::Subnet
  Properties:
    VpcId: !Ref VPC
    AvailabilityZone: !Select [ 1, !GetAZs '' ]
    CidrBlock: 10.0.64.0/18
    MapPublicIpOnLaunch: true
    Tags:
      - Key: Name
        Value: !Ref AWS::StackName
      - Key: AZ
        Value: !Select [ 1, !GetAZs '' ]
```

main.yml

**Line #15 and #28:** `!GetAZs` is a CloudFormation function that returns an array of the available availability zones.

**Line #15 and #28:** `!Select` is a CloudFormation function that pulls an object out of an array by its index.

**Line #30:** We'll need a public IP in order to SSH to the instances. We'll lock this down when we work on network security later.

We also need to add an internet gateway now that we're no longer using the default one. The internet gateway makes it possible for our hosts to route network traffic to and from the internet.

```
InternetGateway:
  Type: AWS::EC2::InternetGateway
  Properties:
    Tags:
      - Key: Name
        Value: !Ref AWS::StackName

InternetGatewayAttachment:
  Type: AWS::EC2::VPCGatewayAttachment
  Properties:
    InternetGatewayId: !Ref InternetGateway
    VpcId: !Ref VPC
```

main.yml

We also need to set up a routing table for our subnets to tell them to use our internet gateway.

```
RouteTable:
  Type: AWS::EC2::RouteTable
  Properties:
    VpcId: !Ref VPC
    Tags:
      - Key: Name
        Value: !Ref AWS::StackName

DefaultPublicRoute:
  Type: AWS::EC2::Route
  DependsOn: InternetGatewayAttachment
  Properties:
    RouteTableId: !Ref RouteTable
    DestinationCidrBlock: 0.0.0.0/0
    GatewayId: !Ref InternetGateway

SubnetRouteTableAssociationAZ1:
  Type: AWS::EC2::SubnetRouteTableAssociation
  Properties:
    RouteTableId: !Ref RouteTable
    SubnetId: !Ref SubnetAZ1

SubnetRouteTableAssociationAZ2:
  Type: AWS::EC2::SubnetRouteTableAssociation
  Properties:
    RouteTableId: !Ref RouteTable
    SubnetId: !Ref SubnetAZ2
```

```
      SubnetId: !Ref SubnetAZ2
```

main.yml

Now it’s time to create the load balancer itself. The load balancer will exist in both of our subnets.

```
LoadBalancer:
  Type: AWS::ElasticLoadBalancingV2::LoadBalancer
  Properties:
    Type: application
    Scheme: internet-facing
    SecurityGroups:
      - !GetAtt SecurityGroup.GroupId
    Subnets:
      - !Ref SubnetAZ1
      - !Ref SubnetAZ2
    Tags:
      - Key: Name
        Value: !Ref AWS::StackName
```

main.yml

Then, let’s configure our load balancer to listen for HTTP traffic on port 80, and forward that traffic to a target group named `LoadBalancerTargetGroup`.

```
LoadBalancerListener:
  Type: AWS::ElasticLoadBalancingV2::Listener
  Properties:
    DefaultActions:
      - Type: forward
        TargetGroupArn: !Ref LoadBalancerTargetGroup
    LoadBalancerArn: !Ref LoadBalancer
    Port: 80
    Protocol: HTTP
```

Now we can create the target group that references our two EC2 instances and the HTTP port they’re listening on.

```
LoadBalancerTargetGroup:
  Type: AWS::ElasticLoadBalancingV2::TargetGroup
  Properties:
    TargetType: instance
    Port: 8080
    Protocol: HTTP
    VpcId: !Ref VPC
    HealthCheckEnabled: true
    HealthCheckProtocol: HTTP
    Targets:
      - Id: !Ref Instance
      - Id: !Ref Instance2
    Tags:
      - Key: Name
        Value: !Ref AWS::StackName
```

main.yml

We will place each instance in one of the subnets we created by adding a `SubnetId` property to each. This will also place each instance into the availability zone specified by the subnet.

```
Instance:
  Type: AWS::EC2::Instance
  CreationPolicy:
    ResourceSignal:
      Timeout: PT5M
      Count: 1
  Properties:
    SubnetId: !Ref SubnetAZ1
    LaunchTemplate:
      LaunchTemplateId: !Ref InstanceLaunchTemplate
      Version: !GetAtt InstanceLaunchTemplate.LatestVersionNumber
    Tags:
      - Key: Name
        Value: !Ref AWS::StackName

Instance2:
  Type: AWS::EC2::Instance
  CreationPolicy:
    ResourceSignal:
      Timeout: PT5M
      Count: 1
  Properties:
    SubnetId: !Ref SubnetAZ2
    LaunchTemplate:
      LaunchTemplateId: !Ref InstanceLaunchTemplate
      Version: !GetAtt InstanceLaunchTemplate.LatestVersionNumber
    Tags:
      - Key: Name
        Value: !Ref AWS::StackName
```

main.yml

**On line #8 and #23:** Puts `Instance` in `SubnetAZ1` and `Instance2` in `SubnetAZ2`.

Now let's return to our security group resource. We'll need to add a reference to the VPC we created. We also need to open port 80, as that's what our load balancer is listening on.

```
SecurityGroup:
  Type: AWS::EC2::SecurityGroup
  Properties:
    VpcId: !Ref VPC
    GroupDescription:
      !Sub 'Internal Security group for ${AWS::StackName}'
    SecurityGroupIngress:
      - IpProtocol: tcp
        FromPort: 8080
        ToPort: 8080
        ToPort: 8080
        CidrIp: 0.0.0.0/0
      - IpProtocol: tcp
        FromPort: 80
        ToPort: 80
        CidrIp: 0.0.0.0/0
      - IpProtocol: tcp
        FromPort: 22
        ToPort: 22
        CidrIp: 0.0.0.0/0
    Tags:
      - Key: Name
        Value: !Ref AWS::StackName
```

main.yml

**Line #4:** References our VPC.

**Line #12:** Adds a new rule to allow internet traffic to port 80.

Finally, let's wrap up these changes by making our CloudFormation template output the domain name of the load balancer instead of the individual EC2 instances.

```
LBEndpoint:
  Description: The DNS name for the LB
  Value: !Sub "http://${LoadBalancer.DNSName}:80"
  Export:
    Name: LBEndpoint
```

main.yml

And let's also modify the `deploy-infra.sh` script to give us the load balancer's endpoint.

```
# If the deploy succeeded, show the DNS name of the created instance
if [ $? -eq 0 ]; then
  aws cloudformation list-exports \
    --profile awsbootstrap \
    --query "Exports[?ends_with(Name,'LBEndpoint')].Value"
fi
```

deploy-infra.sh

**Line #5:** Using `ends_with` here is not necessary right now, but will be useful when we get to the `hello-prod` section.

Now we're ready to deploy our new infrastructure.

```
./deploy-infra.sh
```

```
=========== Deploying setup.yml ===========


Waiting for changeset to be created..

No changes to deploy. Stack awsbootstrap-setup is up to date



=========== Deploying main.yml ===========

Waiting for changeset to be created..
Waiting for stack create/update to complete
Successfully created/updated stack - awsbootstrap
[
    "http://awsbo-LoadB-15405MBPT66BQ-476189870.us-east-1.elb.amazonaws.com:80"
]
```

terminal

If the instances needed to be recreated, then our application won't be running on them automatically. We can trigger the deployment manually by hitting *Release Change* in the CodePipeline console.

Once the deployment is complete, we should be able to reach our application through the load balancer endpoint. If we try to make several requests to this endpoint, we should be able to see the load balancer in action, where we get a different message depending on which of our two instances responded to the request.

```
for run in {1..10}; do curl -s http://awsbo-LoadB-15405MBPT66BQ-476189870.us-east-1.elb.amazonaws.
   4 Hello World from ip-10-0-113-245.ec2.internal
   6 Hello World from ip-10-0-61-251.ec2.internal
```

terminal

Now it's time to push all our infrastructure changes to GitHub and move to the next section.

```
git add main.yml deploy-infra.sh
git commit -m "Add ALB Load Balancer"
git push
```

terminal

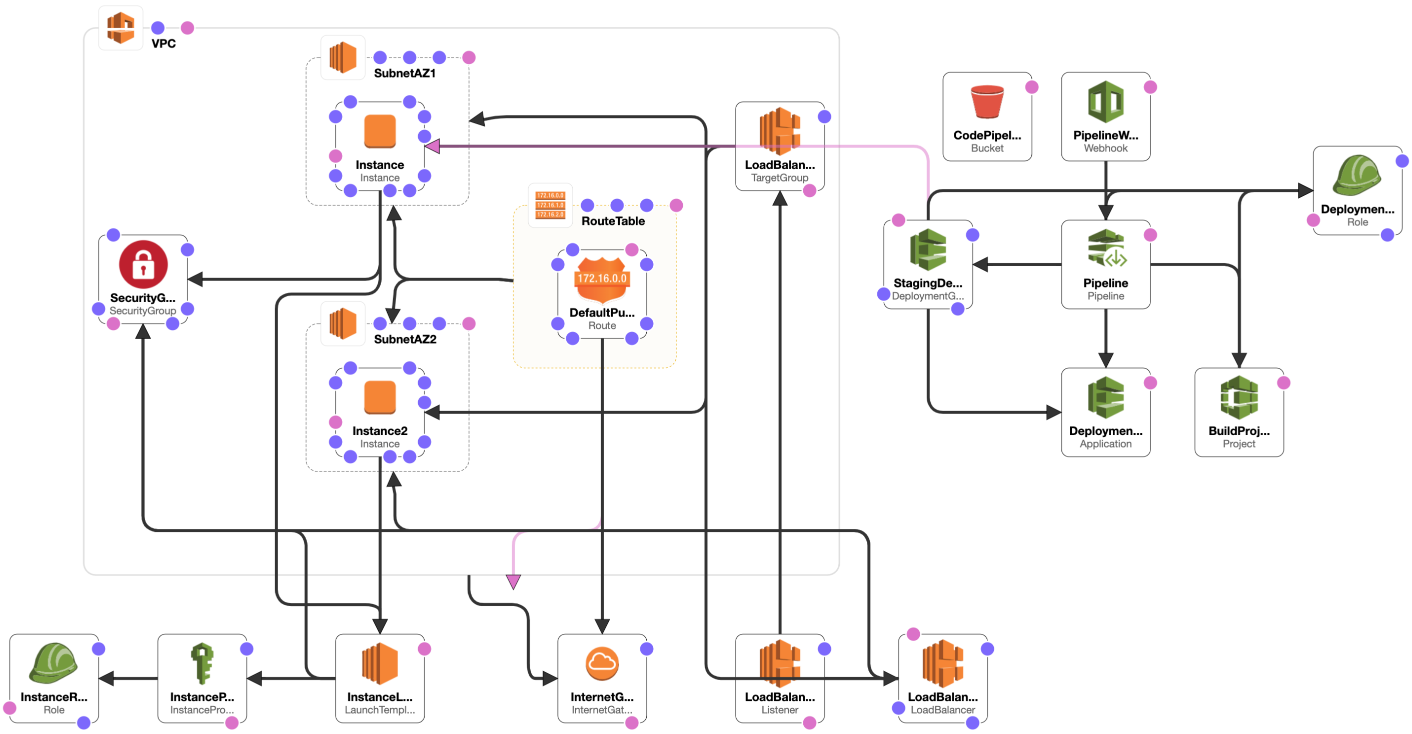
> **Note:** All the code has been already added and we are pushing it on our repository as well.

This code requires the following API keys to execute:

| | |
|---|---|
| username | Not Specified... |
| AWS_ACCESS_KE... | Not Specified... |
| AWS_SECRET_AC... | Not Specified... |
| AWS_REGION | us-east-1 |
| Github_Token | Not Specified... |

```
const { hostname } = require('os');
const http = require('http');
const message = `Hello World from ${hostname()}\n`;
const port = 8080;
const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end(message);
});
  server.listen(port, hostname, () => {
    console.log(`Server running at http://${hostname()}:${port}/`);
});
```

In order to get a pictorial view of our developed cloudformation stack so far, below is the design view which shows the resources we created and their relationships.



Add an Application Load Balancer

In the next lesson, we will replace explicit EC2 instances with Auto Scaling.