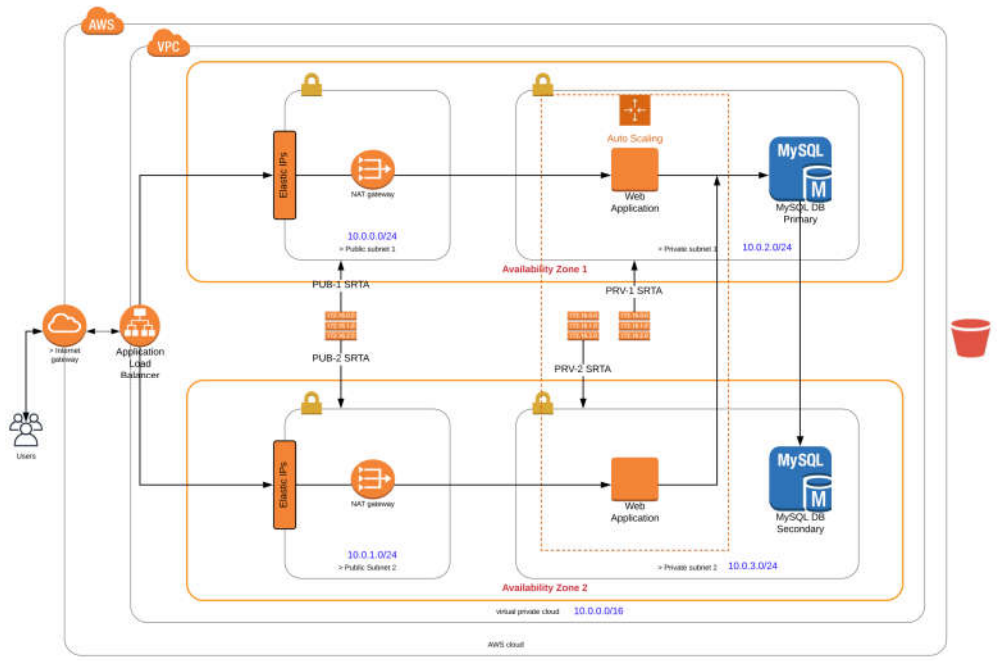
3.C. Networking Infrastructure

- Network Implementation Diagram
    - Adding Subnets
    - Adding a NAT Gateway
    - Routing
        - Route Tables
        - Routes
        - SubnetRouteTableAssociation
    - Outputs
        - Join Function

Network Implementation Diagram



Don’t hard-code parameters

Avoid hard coding parameter values. Instead, use a separate parameter file to store parameter values. Note that the parameter file should be in .json format, as .yml format is not yet supported for the parameter file.

Location Dependencies

- `Parameters` should be declared above your `Resources`.
- CloudFormation knows to create the resources in order, based on their dependencies (VPC and InternetGateway, before creating the InternetGatewayAttachment).

Default Parameters

You can also provide default values for parameters in case one was not passed in. In this example you can see that VpcCIDR has a default value of `10.0.0.0/16`.

```
aws cloudformation create-stack --stack-name MyStack --template-body file://MyCloudformatic
```

Adding Subnets

To specify a `Subnet` for your `VPC` you use the following syntax:

```
Type: AWS::EC2::Subnet
Properties:
  AssignIpv6AddressOnCreation: Boolean
  AvailabilityZone: String
  CidrBlock: String
  Ipv6CidrBlock: String
  MapPublicIpOnLaunch: Boolean
  Tags:
    - Tag
  VpcId: String
```

Here is the actual setup of our 2 private `Subnets`:

```
    PrivateSubnet1:
        Type: AWS::EC2::Subnet
        Properties:
            VpcId: !Ref VPC
            AvailabilityZone: !Select [ 0, !GetAZs '' ]
            CidrBlock: !Ref PrivateSubnet1CIDR
            MapPublicIpOnLaunch: false
            Tags:
                - Key: Name
                  Value: !Sub ${EnvironmentName} Private Subnet (AZ1)

    PrivateSubnet2:
        Type: AWS::EC2::Subnet
        Properties:
            VpcId: !Ref VPC
            AvailabilityZone: !Select [ 1, !GetAZs '' ]
            CidrBlock: !Ref PrivateSubnet2CIDR
            MapPublicIpOnLaunch: false
            Tags:
                - Key: Name
                  Value: !Sub ${EnvironmentName} Private Subnet (AZ2)
```

You can see the index being used from the returning `AvailabilityZone`'s array. Notice that our `subnets` are not sharing `AvailabilityZones`. We are keeping them separated like we displayed in our diagrams from the previous lesson:

PrivateSubnet1: `AvailabilityZone: !Select [ 0, !GetAZ's '' ]`

PrivateSubnet2: `AvailabilityZone: !Select [ 1, !GetAZ's '' ]`

This code:

```
!select [0, !GetAZs‘’]
```

calls the function `GetAZ`, which returns a list of availability zones, which are indexed 0, 1 etc.

Tip
Name your subnets using tags, to keep track when you create many subnets.

Adding a NAT Gateway

You can use `NAT Gateways` in both your public and/or private `Subnets`. The following code is the basic syntax for declaring a `NAT Gateway`:

```yaml
Type: AWS::EC2::NatGateway
Properties:
  AllocationId: String
  SubnetId: String
  Tags:
    - Tag
```

The following declarations are from the sample code shown in the above video:

```yaml
NatGateway1EIP:
    Type: AWS::EC2::EIP
    DependsOn: InternetGatewayAttachment
    Properties:
        Domain: vpc

NatGateway2EIP:
    Type: AWS::EC2::EIP
    DependsOn: InternetGatewayAttachment
    Properties:
        Domain: vpc

NatGateway1:
    Type: AWS::EC2::NatGateway
    Properties:
        AllocationId: !GetAtt NatGateway1EIP.AllocationId
        SubnetId: !Ref PublicSubnet1

NatGateway2:
    Type: AWS::EC2::NatGateway
    Properties:
        AllocationId: !GetAtt NatGateway2EIP.AllocationId
        SubnetId: !Ref PublicSubnet2
```

The `EIP` in `AWS::EC2::EIP`stands for Elastic IP. This will give us a known/constant IP address to use instead of a disposable or ever-changing IP address. This is important when you have applications that depend on a particular IP address. `NatGateway1EIP` uses this type for that very reason:

```yaml
NatGateway1EIP:
    Type: AWS::EC2::EIP
    DependsOn: InternetGatewayAttachment
    Properties:
        Domain: vpc
```

Tip:

- Use the `DependsOn` attribute to protect your dependencies from being created without the proper requirements.
  In the scenario above the `EIP` allocation will only happen after the `InternetGatewayAttachment` has completed.

Routing

Routing: Routing is the action of applying routing rules to your network, in this case, to your VPC.

Routing rule: Resources follow the routing rule, which defines what resource has access to communicate with another resource. It blocks traffic from resources that do not follow the routing rule.

Route Tables

We create `RouteTables` for `VPCs` so that we can add `Routes` that we later associate with `Subnets`. The following is the syntax used to define a `RouteTable`:

```
Type: AWS::EC2::RouteTable
Properties:
  Tags:
    - Tag
  VpcId: String
```

The only required property for setting up a `RouteTable` is the `VpcId`. Here is an example table from the video lesson:

```
PublicRouteTable:
        Type: AWS::EC2::RouteTable
        Properties:
            VpcId: !Ref VPC
            Tags:
                - Key: Name
                  Value: !Sub ${EnvironmentName} Public Routes
```

Routes

The following is the syntax used to set up our `Route`:

```
Type: AWS::EC2::Route
Properties:
  DestinationCidrBlock: String
  DestinationIpv6CidrBlock: String
  EgressOnlyInternetGatewayId: String
  GatewayId: String
  InstanceId: String
  NatGatewayId: String
  NetworkInterfaceId: String
  RouteTableId: String
  VpcPeeringConnectionId: String
```

The `DestinationCidrBlock` property is used for destination matching and a `wildcard address` (`0.0.0.0/0`) to reference all traffic.

So in the following example, when we use the wildcard address `0.0.0.0/0`, we are saying for any address that comes through this route, send it to the referenced `GatewayId`:

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

SubnetRouteTableAssociation

In order to associate `Subnets` with our `Route Table` we will need to use a `SubnetRouteTableAssociation`. `SubnetRouteTableAssociations` are defined using the following syntax:

```
Type: AWS::EC2::SubnetRouteTableAssociation
Properties:
  RouteTableId: String
  SubnetId: String
```

This only takes two properties, which are the id’s used for our `RouteTable` and our `Subnet`. You can see references used in the example from our video lesson above:

```
PublicSubnet1RouteTableAssociation:
        Type: AWS::EC2::SubnetRouteTableAssociation
        Properties:
            RouteTableId: !Ref PublicRouteTable
            SubnetId: !Ref PublicSubnet1
```

Important Note:

- `Routes` should be defined starting with the most specific rule and transitioning to the least specific rule.
- Private-VPC servers are not going to have public IP address.
    - So, even if they are placed on a public subnet there is no way to access them if they have no IP address.
- Having 2 Route-Tables to cover both private-subnets seem unnecessary, however this is a good practice in order to take into account future network expansion.

```
PublicRouteTable:
    Type: AWS::EC2::RouteTable
    Properties:
        VpcId: !Ref VPC
        Tags:
            - Key: Name
              Value: !Sub ${EnvironmentName} Public Routes

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

PublicSubnet1RouteTableAssociation:
    Type: AWS::EC2::SubnetRouteTableAssociation
    Properties:
        RouteTableId: !Ref PublicRouteTable
        SubnetId: !Ref PublicSubnet1

PublicSubnet2RouteTableAssociation:
    Type: AWS::EC2::SubnetRouteTableAssociation
    Properties:
        RouteTableId: !Ref PublicRouteTable
        SubnetId: !Ref PublicSubnet2


PrivateRouteTable1:
    Type: AWS::EC2::RouteTable
    Properties:
        VpcId: !Ref VPC
        Tags:
            - Key: Name
              Value: !Sub ${EnvironmentName} Private Routes (AZ1)

DefaultPrivateRoute1:
    Type: AWS::EC2::Route
    Properties:
        RouteTableId: !Ref PrivateRouteTable1
        DestinationCidrBlock: 0.0.0.0/0
        NatGatewayId: !Ref NatGateway1

PrivateSubnet1RouteTableAssociation:
    Type: AWS::EC2::SubnetRouteTableAssociation
    Properties:
        RouteTableId: !Ref PrivateRouteTable1
        SubnetId: !Ref PrivateSubnet1

PrivateRouteTable2:
    Type: AWS::EC2::RouteTable
    Properties:
        VpcId: !Ref VPC
        Tags:
            - Key: Name
              Value: !Sub ${EnvironmentName} Private Routes (AZ2)

DefaultPrivateRoute2:
    Type: AWS::EC2::Route
    Properties:
        RouteTableId: !Ref PrivateRouteTable2
        DestinationCidrBlock: 0.0.0.0/0
        NatGatewayId: !Ref NatGateway2

PrivateSubnet2RouteTableAssociation:
    Type: AWS::EC2::SubnetRouteTableAssociation
    Properties:
        RouteTableId: !Ref PrivateRouteTable2
        SubnetId: !Ref PrivateSubnet2
```

If your servers have no internet access it’s probably because…

- You created the internet gateway but forgot to attach it to your VPC
- You placed your NAT Gateways inside private subnets with no routes to the outside world
- You have a missing route in your routing table
- You created a routing table but forgot to associate your subnet(s) with it.

Documentation:

- Route Tables Overview
- Route Table Documentation
- Route Documentation
- Subnet Route Table Association Documentation

Outputs

`Outputs` are optional but are very useful if there are output values you need to:

- import into another stack
- return in a response
- view in AWS console

To declare an `Output` use the following syntax:

```
Outputs:
  Logical ID:
    Description: Information about the value
    Value: Value to return
    Export:
      Name: Value to export
```

The `Value` is required but the `Name` is optional. In the following example we are returning the id of our `VPC` as well as our Environment’s Name:

```
VPC:
        Description: A reference to the created VPC
        Value: !Ref VPC
        Export:
          Name: !Sub ${EnvironmentName}-VPCID
```

Join Function

You can use the `join` function to combine a group of `values`. The syntax requires you provide a `delimiter` and a list of values you want appended.

`Join` function syntax:

```
Fn::Join: [ delimiter, [ comma-delimited list of values ] ]
```

In the following example we are using `!Join` to combine our subnets before returning their values:

```
PublicSubnets:
        Description: A list of the public subnets
        Value: !Join [ ",", [ !Ref PublicSubnet1, !Ref PublicSubnet2 ]]
        Export:
          Name: !Sub ${EnvironmentName}-PUB-NETS
```

Documentation:

- Outputs Documentation
- Join Function
- Substitutes

Top

Deleting VPC

Are you sure that you want to delete this VPC (vpc-084aa83cfa00ca370 UdacityProject)?

Deleting this VPC will also delete these objects associated with this VPC in this region:

- Subnets
- Security Groups
- Network ACLs
- Internet Gateways
- Egress Only Internet Gateways
- Route Tables
- Network Interfaces
- Peering Connections
- Endpoints