User Manual

SKIPQ

DEVAESH KAGGDAS

Table of Contents

Project description –

- The purpose of this project was to build and develop a website called EngineRay, a job search and recruitment platform designed for multiple companies and engineers who are job seekers. Engineers can create their own profiles, which will be enriched by the vetting data generated by EngineRay.
- One of the key aspects of this project was to move the system from a traditional server to an AWS based hosting.

Steps to run deploy and run the resources : -

1. To get started install rpm in our cloud environment

npm install -g aws-cdk

2. Create the project directory

mkdir engineray

3. Path to the file

cd eningeray

4. set the programming language for the project

cdk init sample-app --language typescript

5. Clone existing repo from GitHub

git clone "GitHub http path"

Note - The 'ts.code' file in the bin folder is the starting point of the code, any new stacks created in the future must be first imported here.

Useful commands

'cdk ls' - to list the stacks

'cdk synth EngineRayProect' - synthesize the stack

'cdk deploy EngineRayProject' - deplpy the stack

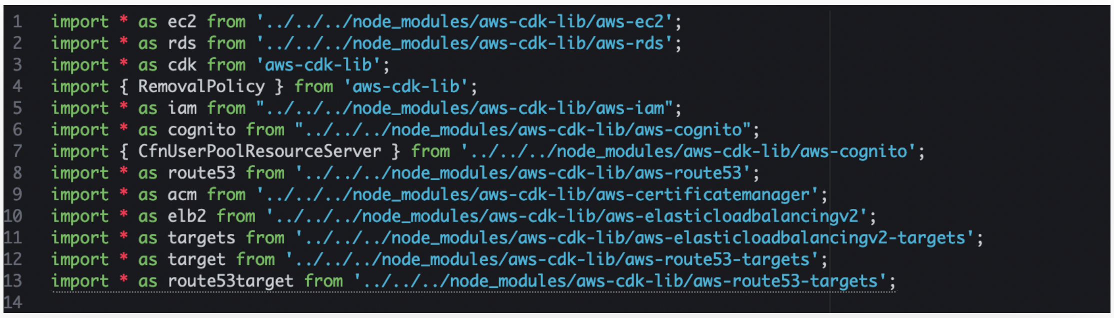
'cdk diff` - compare deployed stack with current state

Code In detail -

The code used to build and deploy the resources mentioned in the Architectural Design is discussed in detail in the following sections.

EngineRayProject Stack Libraries -

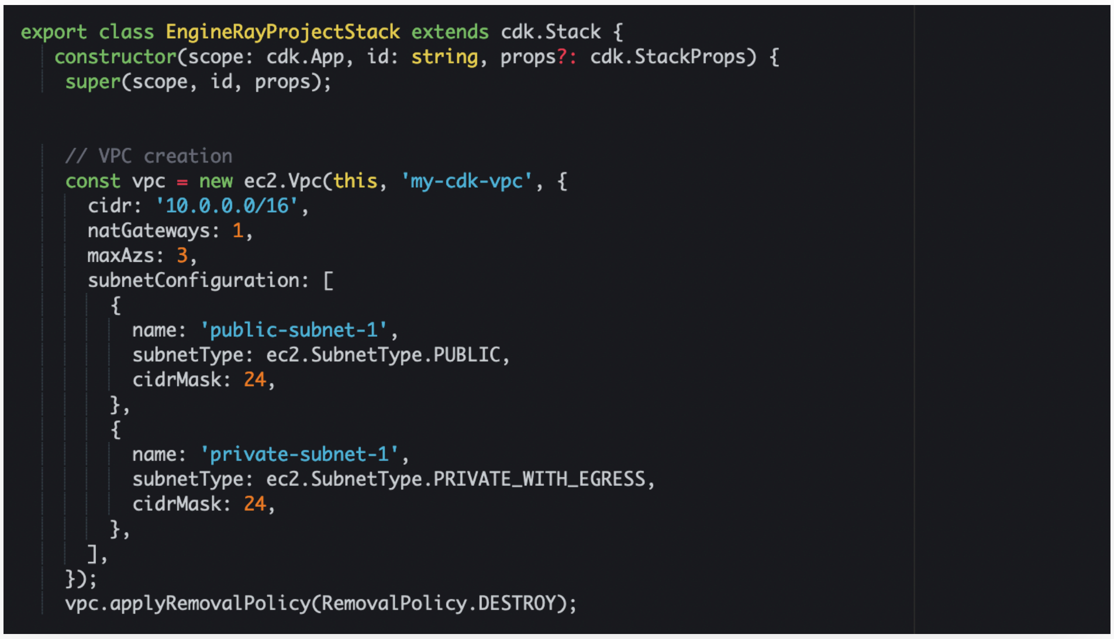
The service names utilized to import various AWS Construct Library modules into our code are as follows.



```
import * as ec2 from '../../../node_modules/aws-cdk-lib/aws-ec2';
import * as rds from '../../../node_modules/aws-cdk-lib/aws-rds';
import * as cdk from 'aws-cdk-lib';
import { RemovalPolicy } from 'aws-cdk-lib';
import * as iam from "../../../node_modules/aws-cdk-lib/aws-iam";
import * as cognito from "../../../node_modules/aws-cdk-lib/aws-cognito";
import { CfnUserPoolResourceServer } from '../../../node_modules/aws-cdk-lib/aws-cognito';
import * as route53 from '../../../node_modules/aws-cdk-lib/aws-route53';
import * as acm from '../../../node_modules/aws-cdk-lib/aws-certificatemanager';
import * as elb2 from '../../../node_modules/aws-cdk-lib/aws-elasticloadbalancingv2';
import * as targets from '../../../node_modules/aws-cdk-lib/aws-elasticloadbalancingv2-targets';
import * as target from '../../../node_modules/aws-cdk-lib/aws-route53-targets';
import * as route53target from '../../../node_modules/aws-cdk-lib/aws-route53-targets';
```

VPC Creation –

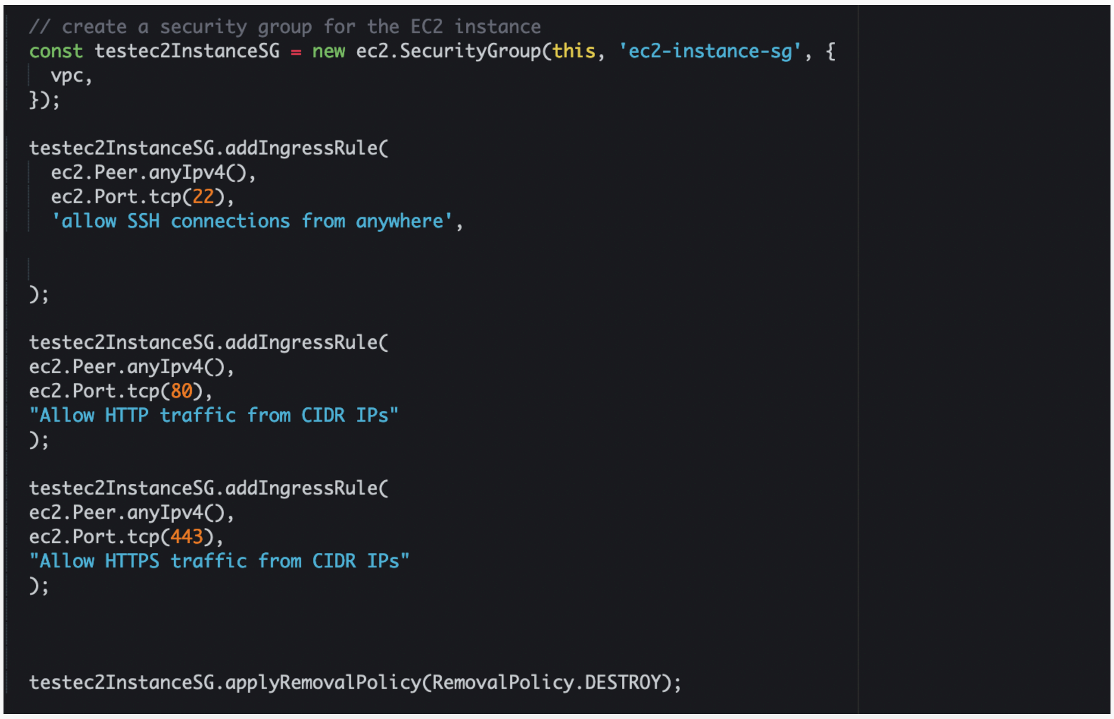
The code below is written to build a VPC with two primary subnets, one public and one private, using NAT Gateway. The CIDR of the VPC is '10.0.0.0/16' with a cidrMask of 24 and three availability zones. The VPC also contains a removal policy, which states that if the stack is destroyed, the VPC will be deleted.



```
export class EngineRayProjectStack extends cdk.Stack {
  constructor(scope: cdk.App, id: string, props?: cdk.StackProps) {
   super(scope, id, props);


   // VPC creation
   const vpc = new ec2.Vpc(this, 'my-cdk-vpc', {
     cidr: '10.0.0.0/16',
     natGateways: 1,
     maxAzs: 3,
     subnetConfiguration: [
       {
         name: 'public-subnet-1',
         subnetType: ec2.SubnetType.PUBLIC,
         cidrMask: 24,
       },
       {
         name: 'private-subnet-1',
         subnetType: ec2.SubnetType.PRIVATE_WITH_EGRESS,
         cidrMask: 24,
       },
     ],
   });
   vpc.applyRemovalPolicy(RemovalPolicy.DESTROY);
```

EC2 Instance Security Group -

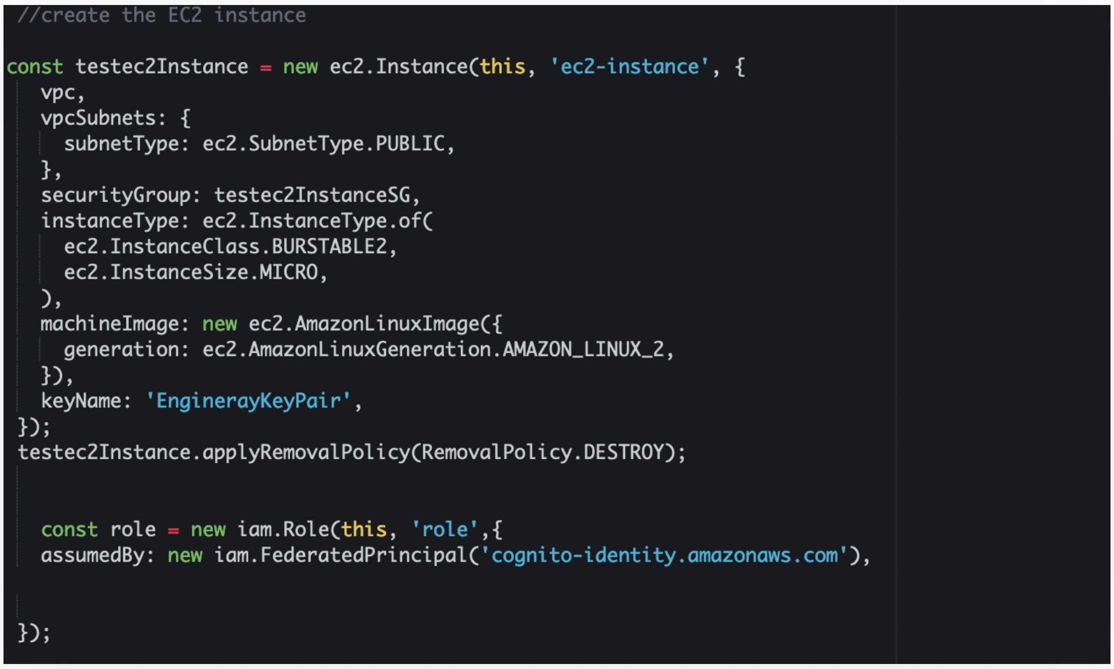
The code for configuring the security group for the ec2 instance is shown below. The security group is allocated to the previously defined VPC. This security group has three key inbound rules that enable SSH, HTTP, and HTTPS traffic from any IP address on ports 22, 80, and 443. The ec2 instance security group also includes a removal policy that deletes it when the stack is destroyed.



```
// create a security group for the EC2 instance
const testec2InstanceSG = new ec2.SecurityGroup(this, 'ec2-instance-sg', {
  vpc,
});

testec2InstanceSG.addIngressRule(
  ec2.Peer.anyIpv4(),
  ec2.Port.tcp(22),
  'allow SSH connections from anywhere',


);

testec2InstanceSG.addIngressRule(
ec2.Peer.anyIpv4(),
ec2.Port.tcp(80),
"Allow HTTP traffic from CIDR IPs"
);

testec2InstanceSG.addIngressRule(
ec2.Peer.anyIpv4(),
ec2.Port.tcp(443),
"Allow HTTPS traffic from CIDR IPs"
);



testec2InstanceSG.applyRemovalPolicy(RemovalPolicy.DESTROY);
```

EC2 Instance –

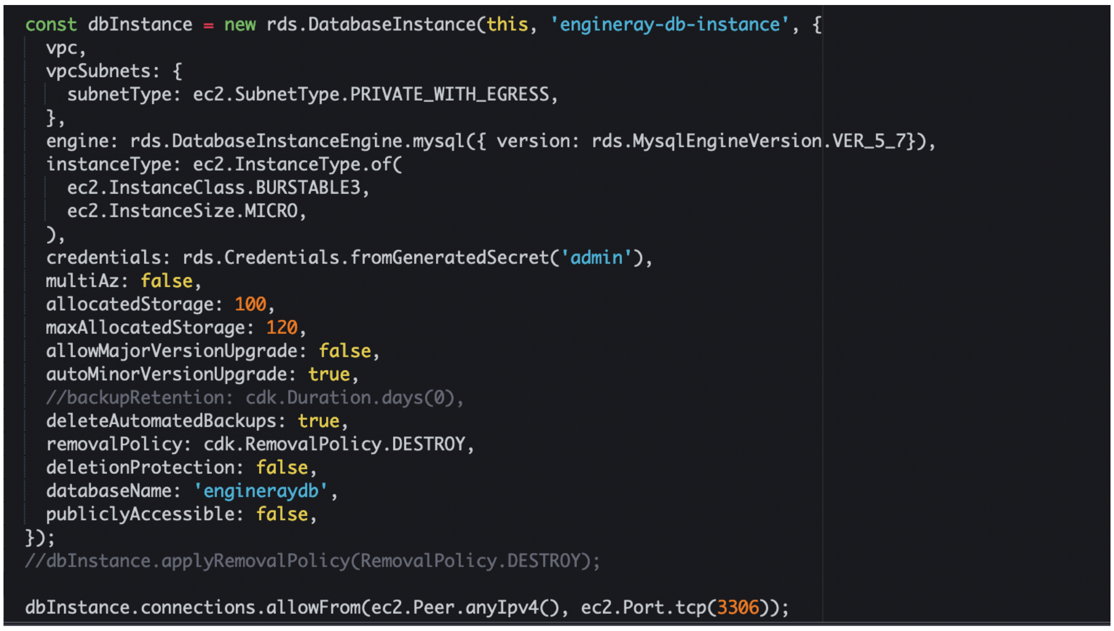
The code below is written to launch an ec2 instance in the public subnet of the previously created VPC. The instance is given the security group 'testec2InstanceSG', which was previously formed. The instance type t2 micro is chosen, with the key name 'EngineraryKeyPair'. In addition, the ec2 instance includes a removal policy that deletes the instance upon stack destroy.



```
//create the EC2 instance

const testec2Instance = new ec2.Instance(this, 'ec2-instance', {
   vpc,
   vpcSubnets: {
     subnetType: ec2.SubnetType.PUBLIC,
   },
   securityGroup: testec2InstanceSG,
   instanceType: ec2.InstanceType.of(
     ec2.InstanceClass.BURSTABLE2,
     ec2.InstanceSize.MICRO,
   ),
   machineImage: new ec2.AmazonLinuxImage({
     generation: ec2.AmazonLinuxGeneration.AMAZON_LINUX_2,
   }),
   keyName: 'EngineraykeyPair',
 });
 testec2Instance.applyRemovalPolicy(RemovalPolicy.DESTROY);


   const role = new iam.Role(this, 'role',{
   assumedBy: new iam.FederatedPrincipal('cognito-identity.amazonaws.com'),


 });
```

RDS database –

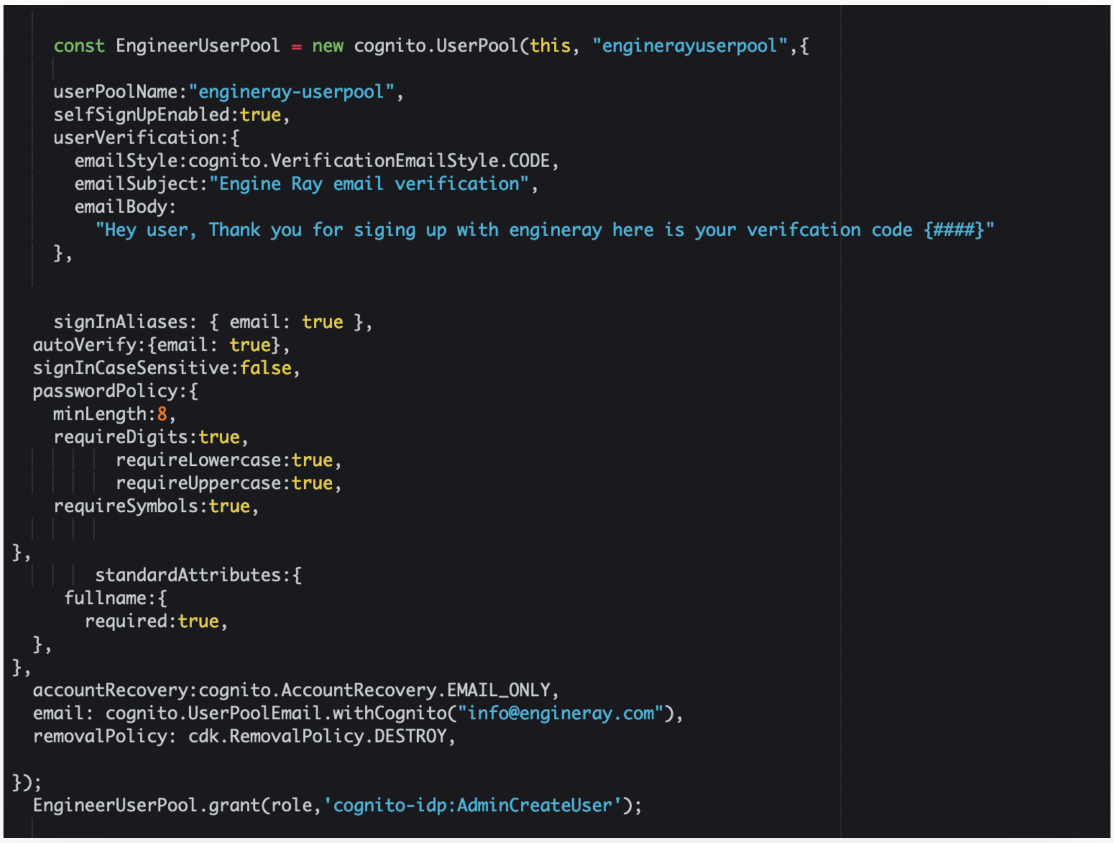
The code below creates an RDS instance with the name 'engineray-db-instance' in the VPC's private subnet. We specify the database type to 'MYSQL' and use MYSQL version 5_7. We utilize AWS Secrets Manager to automatically generate a password for our database using the username 'admin' in order to configure the password for the RDS database. We also limit the maximum storage capacity of the database to 120GB and make it inaccessible to the public in order to prevent unwanted access to and alteration of data contained in the RDS database. The line of code 'dbinstance.connections.allowFrom(testec2Instance, ec2.Port.tcp(3306))' is written to allow the ec2 instance created previously to connect to the RDS instance; as a result, the webpages hosted in the ec2 may now communicate with the RDS database.



```
const dbInstance = new rds.DatabaseInstance(this, 'engineray-db-instance', {
  vpc,
  vpcSubnets: {
    subnetType: ec2.SubnetType.PRIVATE_WITH_EGRESS,
  },
  engine: rds.DatabaseInstanceEngine.mysql({ version: rds.MysqlEngineVersion.VER_5_7}),
  instanceType: ec2.InstanceType.of(
    ec2.InstanceClass.BURSTABLE3,
    ec2.InstanceSize.MICRO,
  ),
  credentials: rds.Credentials.fromGeneratedSecret('admin'),
  multiAz: false,
  allocatedStorage: 100,
  maxAllocatedStorage: 120,
  allowMajorVersionUpgrade: false,
  autoMinorVersionUpgrade: true,
  //backupRetention: cdk.Duration.days(0),
  deleteAutomatedBackups: true,
  removalPolicy: cdk.RemovalPolicy.DESTROY,
  deletionProtection: false,
  databaseName: 'engineraydb',
  publiclyAccessible: false,
});
//dbInstance.applyRemovalPolicy(RemovalPolicy.DESTROY);

dbInstance.connections.allowFrom(ec2.Peer.anyIpv4(), ec2.Port.tcp(3306));
```

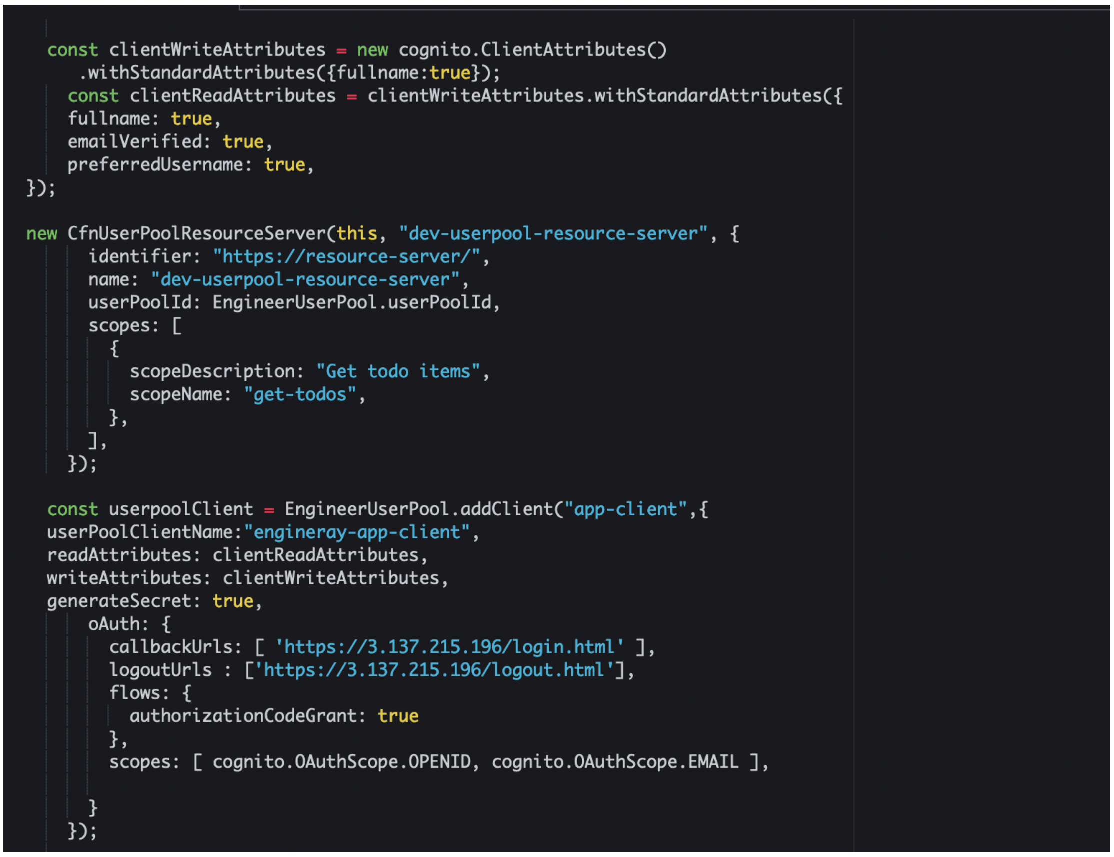
Cognito User Pool –

The code below creates a Cognito UserPool that can be used to sign up/sign in engineers and company users. We create a UserPool called 'enginerayuserpool' and set the mode of verification to email. The user must enter their email address and a password that is at least 8 characters long and includes lowercase, uppercase, and numerals to sign up or register as a user. The user will receive the verification code by email, and after entering it, the user is registered and can use the email id and password to log into the system in the future. The UserPool additionally contains a removal policy that deletes the user pool when the cdk destroy command is used.



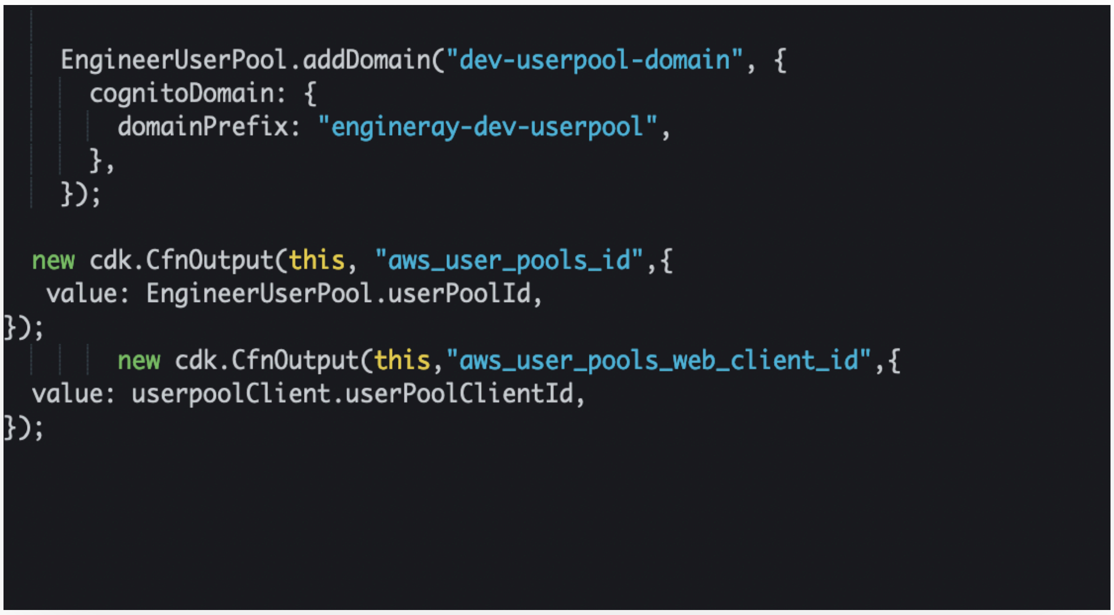
```
  const EngineerUserPool = new cognito.UserPool(this, "enginerayuserpool",{

  userPoolName:"engineray-userpool",
  selfSignUpEnabled:true,
  userVerification:{
    emailStyle:cognito.VerificationEmailStyle.CODE,
    emailSubject:"Engine Ray email verification",
    emailBody:
      "Hey user, Thank you for siging up with engineray here is your verifcation code {####}"
  },


  signInAliases: { email: true },
 autoVerify:{email: true},
 signInCaseSensitive:false,
 passwordPolicy:{
   minLength:8,
   requireDigits:true,
         requireLowercase:true,
         requireUppercase:true,
   requireSymbols:true,

},
     standardAttributes:{
    fullname:{
      required:true,
  },
},
  accountRecovery:cognito.AccountRecovery.EMAIL_ONLY,
  email: cognito.UserPoolEmail.withCognito("info@engineray.com"),
  removalPolicy: cdk.RemovalPolicy.DESTROY,

});
  EngineerUserPool.grant(role,'cognito-idp:AdminCreateUser');
```

The UserPoolResourceServer and UserPoolClient are created next. The resource server can be used to authenticate the user using OAuth and limit the user's access to specified webpages. The UserPoolClient is designed to incorporate the cognito into the produced webpages. We set the UserPoolClient name to 'app-client' and offer OAuth, which prompts users to enter their email id and password, and the scope attribute defines the login and logout urls, where we provide the https address of the login and logout webpages saved in the ec2 instance.



```
  const clientWriteAttributes = new cognito.ClientAttributes()
     .withStandardAttributes({fullname:true});
    const clientReadAttributes = clientWriteAttributes.withStandardAttributes({
    fullname: true,
    emailVerified: true,
    preferredUsername: true,
});

new CfnUserPoolResourceServer(this, "dev-userpool-resource-server", {
      identifier: "https://resource-server/",
      name: "dev-userpool-resource-server",
      userPoolId: EngineerUserPool.userPoolId,
      scopes: [
        {
          scopeDescription: "Get todo items",
          scopeName: "get-todos",
        },
      ],
    });

  const userpoolClient = EngineerUserPool.addClient("app-client",{
  userPoolClientName:"engineray-app-client",
  readAttributes: clientReadAttributes,
  writeAttributes: clientWriteAttributes,
  generateSecret: true,
      oAuth: {
        callbackUrls: [ 'https://3.137.215.196/login.html' ],
        logoutUrls : ['https://3.137.215.196/logout.html'],
        flows: {
          authorizationCodeGrant: true
        },
        scopes: [ cognito.OAuthScope.OPENID, cognito.OAuthScope.EMAIL ],

      }
    });
```

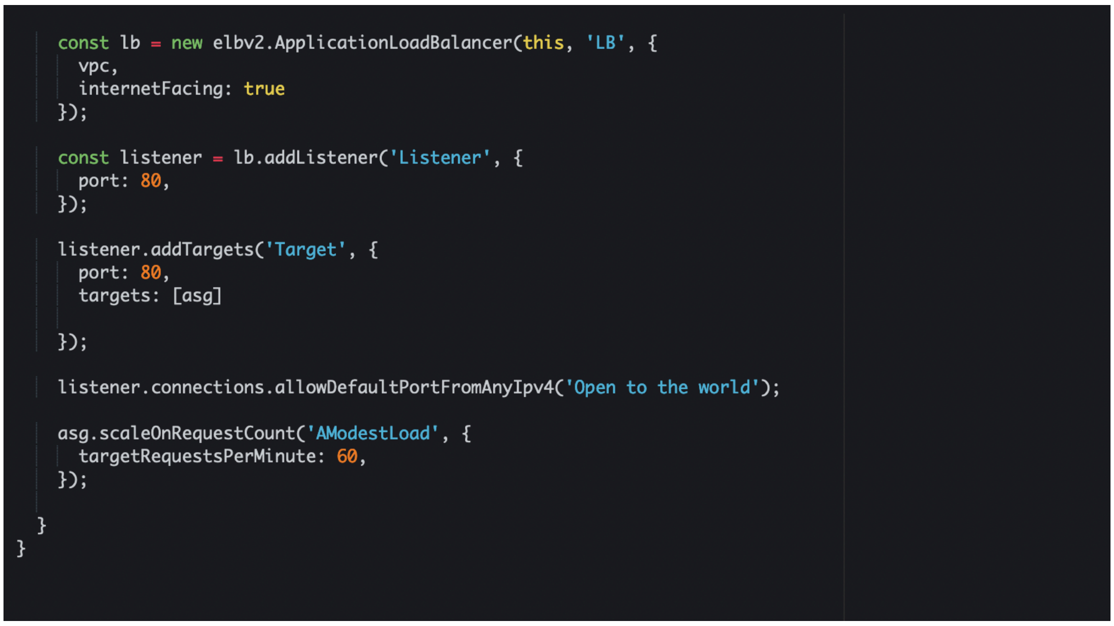
In this section, we specify a domain for UserPool that will serve as the cognito address, and we must ensure that this domain address is globally unique. Then, using the cdk.CfnOutput commands, we output our UserPool and UserPoolClient ids.



```
  EngineerUserPool.addDomain("dev-userpool-domain", {
    cognitoDomain: {
      domainPrefix: "engineray-dev-userpool",
    },
  });

new cdk.CfnOutput(this, "aws_user_pools_id",{
  value: EngineerUserPool.userPoolId,
});
      new cdk.CfnOutput(this,"aws_user_pools_web_client_id",{
  value: userpoolClient.userPoolClientId,
});
```

Load Balancer with target and listener -

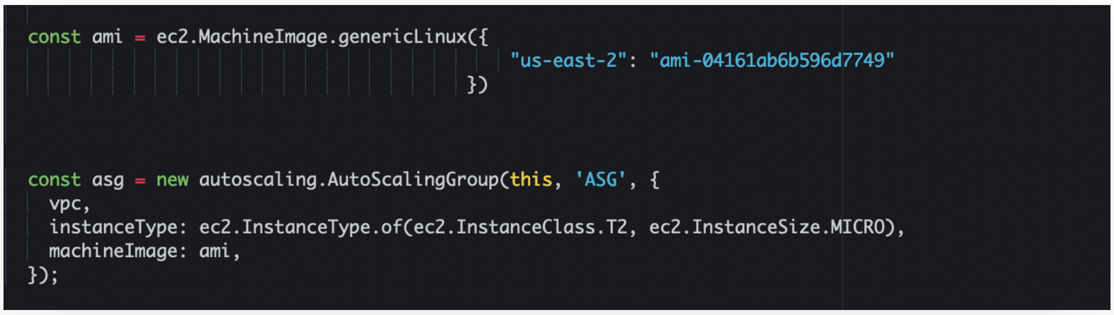
The code below is created to establish a load balancer named 'LB'. We assign the previously created vpc to this load balancer and enable internet facing so that the load balancer may connect to the internet. We add a listener to the loadbalancer on port 80 with the line 'lb.addListener(listener)' and also set the target to port 80 and the target group to the AutoScalingGroup 'asg', which will be constructed in the section that follows. We also use "listener.connections.allowDefaultPortFromAnyIpv4('Open to the World')" to make the loadbalancer available from any IP address.



```
    const lb = new elbv2.ApplicationLoadBalancer(this, 'LB', {
      vpc,
      internetFacing: true
    });

    const listener = lb.addListener('Listener', {
      port: 80,
    });

    listener.addTargets('Target', {
      port: 80,
      targets: [asg]

    });

    listener.connections.allowDefaultPortFromAnyIpv4('Open to the world');

    asg.scaleOnRequestCount('AModestLoad', {
      targetRequestsPerMinute: 60,
    });

  }
}
```

Custom AMI and AutoScailing Group

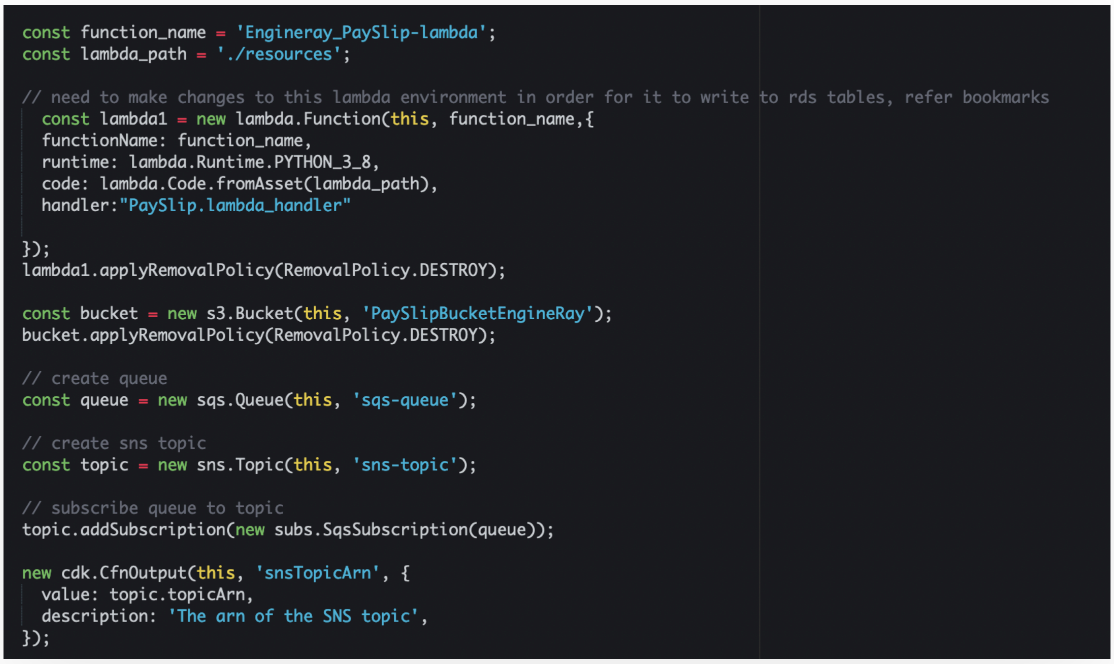
In this code, we first generate an AMI image of the previously created testec2instance using the image's region and ami id. Using this AMI, we launch an AutoScailingGroup and assign it to the VPC we created previously, as well as set the ec2 details to be the same as the ec2 we generated earlier and match the AMI.



```
const ami = ec2.MachineImage.genericLinux({
                                    "us-east-2": "ami-04161ab6b596d7749"
                                })


const asg = new autoscaling.AutoScalingGroup(this, 'ASG', {
  vpc,
  instanceType: ec2.InstanceType.of(ec2.InstanceClass.T2, ec2.InstanceSize.MICRO),
  machineImage: ami,
});
```

SNS and SQS Queue with lambda –

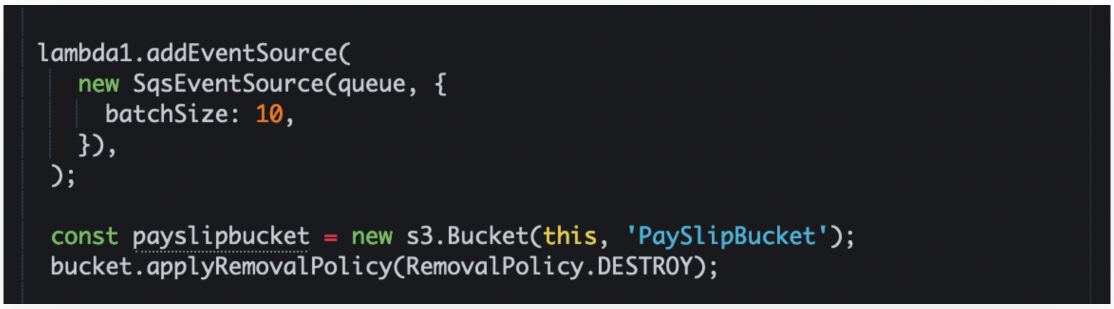
n this code, we define the name of the lambda function as well as the path to the lambda function. As the lambda function will be written in Python, we set the runtime to 'PYTHON_3_8'. The lambda also contains a removal policy, which deletes the lambda function when the cdk destroy command is used.   Then we use the "new s3.Bucket(this, 'PaySlipBucketEngineRay')" line to create an S3 bucket. Then we build the'sns-queue' and'sns-topic' and make the queue subscribe to the SNS topic using the 'topic.addSubscription(new subs.SqsSubscription(queue))' line of code. Finally we use the cdk.CfnOutput command to get the ARN of the SNS topic.



```
const function_name = 'Engineray_PaySlip-lambda';
const lambda_path = './resources';

// need to make changes to this lambda environment in order for it to write to rds tables, refer bookmarks
  const lambda1 = new lambda.Function(this, function_name,{
  functionName: function_name,
  runtime: lambda.Runtime.PYTHON_3_8,
  code: lambda.Code.fromAsset(lambda_path),
  handler:"PaySlip.lambda_handler"

});
lambda1.applyRemovalPolicy(RemovalPolicy.DESTROY);

const bucket = new s3.Bucket(this, 'PaySlipBucketEngineRay');
bucket.applyRemovalPolicy(RemovalPolicy.DESTROY);

// create queue
const queue = new sqs.Queue(this, 'sqs-queue');

// create sns topic
const topic = new sns.Topic(this, 'sns-topic');

// subscribe queue to topic
topic.addSubscription(new subs.SqsSubscription(queue));

new cdk.CfnOutput(this, 'snsTopicArn', {
  value: topic.topicArn,
  description: 'The arn of the SNS topic',
});
```

SNS queue added as trigger for lambda and Creation of PaySlip S3 Bucket -

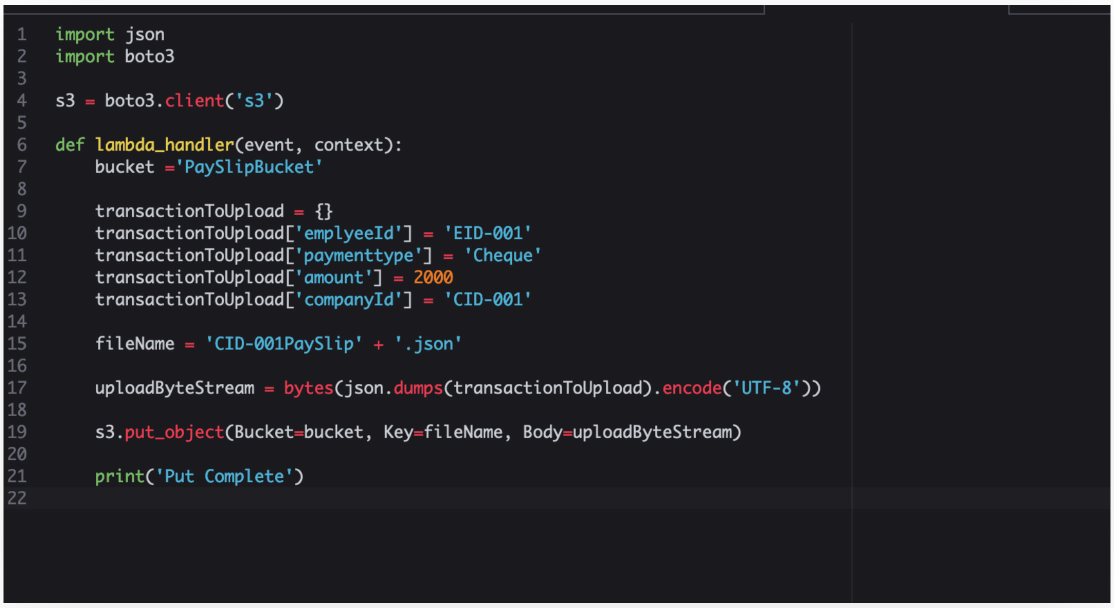
We created the SNS queue and topic in the previous section but did not make them trigger the lambda code. To add the SQS queue as a trigger to the lambda function, we use the 'addEventSource' line.



```
lambda1.addEventSource(
    new SqsEventSource(queue, {
      batchSize: 10,
    }),
  );

  const payslipbucket = new s3.Bucket(this, 'PaySlipBucket');
  bucket.applyRemovalPolicy(RemovalPolicy.DESTROY);
```

PaySlipLambda function

The code for the previously constructed 'PaySlip.py' lambda function is shown below. This is a code that was built to insert payslips that would be generated in the future between the Company and Engineers once an employee has been accepted and begins working for the company. This lambda function writes data to the S3 bucket 'PaySlipBucket'. We generate an employeeId, a paymenttype, an amount, and a companyid, then set their values in a 'json' file and insert it into the specified bucket.



```python
import json
import boto3

s3 = boto3.client('s3')

def lambda_handler(event, context):
    bucket ='PaySlipBucket'

    transactionToUpload = {}
    transactionToUpload['emplyeeId'] = 'EID-001'
    transactionToUpload['paymenttype'] = 'Cheque'
    transactionToUpload['amount'] = 2000
    transactionToUpload['companyId'] = 'CID-001'

    fileName = 'CID-001PaySlip' + '.json'

    uploadByteStream = bytes(json.dumps(transactionToUpload).encode('UTF-8'))

    s3.put_object(Bucket=bucket, Key=fileName, Body=uploadByteStream)

    print('Put Complete')

```