CDK con RDS

David Londoño Palacio

Repositorio (Rama Main): https://github.com/DavidLondo/IaC-CDK-Python

Sección 1: Instalación del entorno local para AWS CDK

1. Instalar Node.js y npm

AWS CDK requiere Node.js.

Descargar Node.js desde la página oficial.

- Instalar siguiendo los pasos del instalador para tu sistema operativo.
- Verificar instalación:

```
node -v
npm -v
```

2. Instalar AWS CLI

AWS CLI permite interactuar con AWS desde la terminal.

Descargar e instalar:

```
curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o "awscliv2.zip"
unzip awscliv2.zip
sudo ./aws/install
```

Verificar la instalación:

```
aws --version
```

3. Configurar credenciales de AWS

AWS CLI necesita tus credenciales:

```
aws configure
```

Se te pedirá:

- AWS Access Key ID
- AWS Secret Access Key
- Default region name (ej: us-east-1)
- Default output format (ej: json)

Estas credenciales se usarán para que CDK despliegue recursos en tu cuenta AWS.

4. Instalar AWS CDK

Instala CDK globalmente usando npm:

```
npm install -g aws-cdk
cdk --version
```

Sección 2: Crear el proyecto AWS CDK

1. Crear la carpeta del proyecto

Elige una ubicación en tu PC y crea una carpeta para tu proyecto CDK:

```
mkdir MyCDKProject
cd MyCDKProject
```

2. Inicializar un proyecto CDK

Decide en qué lenguaje quieres escribir tu stack. Vamos a usar **Python** (puedes usar TypeScript o JavaScript si prefieres).

```
cdk init app --language python
```

Esto hará varias cosas:

- Crea un entorno virtual de Python (`.venv/`).
- Crea archivos base: `app.py` , `requirements.txt` , `cdk.json` .
- Crea una carpeta `MyCDKProject/` donde estarán tus stacks.

3. Activar el entorno virtual

Es importante trabajar dentro del entorno virtual para instalar dependencias:

```
source .venv/bin/activate
```

4. Instalar dependencias necesarias

Dentro del virtualenv, instala AWS CDK para Python y otras librerías necesarias:

```
pip install aws-cdk-lib constructs
```

Sección 3: Definición de componentes

1. Definición de la VPC para el CMS

Primero, es necesario definir el perímetro de la infraestructura, la VPC, en ella se soportará toda da nube que vamos a crear.

Antes de empezar a codificar es necesario ir al archivo cdk.json y añadir lo siguiente al final de context, modificando la parte de las KeyName, que serían las personales de AWS:

```
{
    "vpcCidr": "172.16.0.0/16",
    "maxAzs": 2,
    "natGateways": 2,
    "bastionInstanceType": "t2.micro",
    "bastionKeyName": "si3006",
    "bastionRoleName": "LabRole",
    "bastionVolumeSize": 8,
    "bastionVolumeType": "gp2",
    "bastionAmiParameter": "/aws/service/canonical/ubuntu/server/22.04/stable/current/amd64/hvm/ebs-gp2/ami-id",
    "dbInstanceType": "t2.micro",
    "dbKeyName": "si3006",
    "dbRoleName": "LabRole",
    "dbVolumeSize": 8,
    "dbVolumeType": "gp2",
    "dbAmiParameter": "/aws/service/canonical/ubuntu/server/22.04/stable/current/amd64/hvm/ebs-gp2/ami-id",
    "asgInstanceType": "t2.micro",
    "asgRoleName": "LabRole",
    "asgMinCapacity": 2,
    "asgMaxCapacity": 4,
    "asgHealthGraceMinutes": 3,
    "asgAmiId": "ami-0c02fb55956c7d316",
    "bastionSshPort": 22,
    "dbPort": 3306,
    "dbSshPort": 22,
    "dbAz1Cidr": "172.16.2.0/24",
    "dbAz2Cidr": "172.16.5.0/24",
    "bastionCidr": "172.16.1.0/24",
    "lbHttpPort": 80,
    "cmsHttpPort": 80,
    "cmsSshPort": 22,
    "cmsAz1Cidr": "172.16.1.0/24",
    "cmsAz2Cidr": "172.16.4.0/24"
}
```

Esto con la finalidad de paremetrizar las variables del proyecto, mejorando la seguridad y la modularidad.

Ahora, creamos una carpeta dentro de la del proyecto llamada network, en ella creamos un archivo llamado vpc.py. Dentro ponemos:

```
from aws_cdk import aws_ec2 as ec2, CfnParameter
from constructs import Construct
from aws_cdk import Stack


class CmsVpc(ec2.Vpc):
    def __init__(self, scope: Construct, id: str, **kwargs) → None:
        vpc_cidr = scope.node.try_get_context("vpcCidr") or "172.16.0.0/16"
        max_azs = int(scope.node.try_get_context("maxAzs") or 2)
        nat_gateways = int(scope.node.try_get_context("natGateways") or 2)

        super().__init__(
            scope,
            id,
            ip_addresses=ec2.IpAddresses.cidr(vpc_cidr),
            max_azs=max_azs,
            subnet_configuration=[
                ec2.SubnetConfiguration(
                    name="PublicSubnet",
                    subnet_type=ec2.SubnetType.PUBLIC,
                    cidr_mask=24,
                    map_public_ip_on_launch=True,
                ),
                ec2.SubnetConfiguration(
                    name="PrivateSubnet",
                    subnet_type=ec2.SubnetType.PRIVATE_WITH_EGRESS,
                    cidr_mask=24,
                ),
                ec2.SubnetConfiguration(
                    name="IsolatedSubnet",
                    subnet_type=ec2.SubnetType.PRIVATE_ISOLATED,
                    cidr_mask=24,
                ),
            ],
            nat_gateways=nat_gateways,
            restrict_default_security_group=False,
            **kwargs
        )
        self._assign_explicit_cidrs()

    def _assign_explicit_cidrs(self):
        cidrs = {
            'PublicSubnet': ['172.16.1.0/24', '172.16.4.0/24'],
            'PrivateSubnet': ['172.16.2.0/24', '172.16.5.0/24'],
            'IsolatedSubnet': ['172.16.3.0/24', '172.16.6.0/24']
        }

        for subnet in self.public_subnets:
            subnet.node.add_metadata("CidrBlock", cidrs['PublicSubnet'].pop
(0))
        for subnet in self.private_subnets:
            subnet.node.add_metadata("CidrBlock", cidrs['PrivateSubnet'].pop
(0))
        for subnet in self.isolated_subnets:
            subnet.node.add_metadata("CidrBlock", cidrs['IsolatedSubnet'].pop
(0))
```

Esto crea la VPC con sus subnets.

2. Configuración de los Security Groups

De la misma forma creamos un archivo llamado security_groups.py, y en el ponemos:

```
from aws_cdk import aws_ec2 as ec2
from constructs import Construct



class BastionHostSG(ec2.SecurityGroup):
    def __init__(self, scope: Construct, id: str, vpc: ec2.IVpc, **kwargs):
        super().__init__(
            scope,
            id,
            vpc=vpc,
            description="Enable SSH Access",
            **kwargs
        )

        ssh_port = int(scope.node.try_get_context("bastionSshPort") or 22)

        self.add_ingress_rule(
            peer=ec2.Peer.any_ipv4(),
            connection=ec2.Port.tcp(ssh_port),
            description=f"Allow SSH traffic on port {ssh_port}"
        )


class DatabaseSG(ec2.SecurityGroup):
    def __init__(self, scope: Construct, id: str, vpc: ec2.IVpc, **kwargs):
        super().__init__(
            scope,
            id,
            vpc=vpc,
            description="Allow SQL Access",
            **kwargs
        )

        mysql_port = int(scope.node.try_get_context("dbPort") or 3306)
        ssh_port = int(scope.node.try_get_context("dbSshPort") or 22)

        az1_cidr = scope.node.try_get_context("dbAz1Cidr") or "172.16.2.0/24"
        az2_cidr = scope.node.try_get_context("dbAz2Cidr") or "172.16.5.0/24"
        bastion_cidr = scope.node.try_get_context("bastionCidr") or "172.16.1.0/24"

        self.add_ingress_rule(
            peer=ec2.Peer.ipv4(az1_cidr),
            connection=ec2.Port.tcp(mysql_port),
            description=f"Allow MySQL from CMS AZ1 ({az1_cidr})"
        )
        self.add_ingress_rule(
            peer=ec2.Peer.ipv4(az2_cidr),
            connection=ec2.Port.tcp(mysql_port),
            description=f"Allow MySQL from CMS AZ2 ({az2_cidr})"
        )
        self.add_ingress_rule(
            peer=ec2.Peer.ipv4(bastion_cidr),
            connection=ec2.Port.tcp(ssh_port),
            description=f"Allow SSH from Bastion ({bastion_cidr})"
        )


class LoadBalancerSG(ec2.SecurityGroup):
    def __init__(self, scope: Construct, id: str, vpc: ec2.IVpc, **kwargs):
        super().__init__(
            scope,
            id,
            vpc=vpc,
            description="Allow HTTP/HTTPS to ALB",
            **kwargs
        )

        http_port = int(scope.node.try_get_context("lbHttpPort") or 80)

        self.add_ingress_rule(
            peer=ec2.Peer.any_ipv4(),
            connection=ec2.Port.tcp(http_port),
            description="Allow HTTP from anywhere"
        )
        self.add_ingress_rule(
            peer=ec2.Peer.any_ipv6(),
            connection=ec2.Port.tcp(http_port),
            description="Allow HTTP from IPv6"
        )


class CmsSecurityGroups(Construct):
    def __init__(self, scope: Construct, id: str, vpc: ec2.IVpc, **kwargs):
        super().__init__(scope, id, **kwargs)

        cms_http_port = int(scope.node.try_get_context("cmsHttpPort") or 80)
        cms_ssh_port = int(scope.node.try_get_context("cmsSshPort") or 22)

        cms_az1_cidr = scope.node.try_get_context("cmsAz1Cidr") or "172.16.1.
0/24"
        cms_az2_cidr = scope.node.try_get_context("cmsAz2Cidr") or "172.16.
4.0/24"
        bastion_cidr = scope.node.try_get_context("bastionCidr") or "172.16.1.
0/24"

        self.cms_sg = ec2.SecurityGroup(
            self, "WebCMS",
            vpc=vpc,
            description="Enable HTTP Access",
            allow_all_outbound=True
        )
        self.cms_sg.add_ingress_rule(
            peer=ec2.Peer.ipv4(cms_az1_cidr),
            connection=ec2.Port.tcp(cms_http_port),
            description=f"Permit Web Requests from AZ1 ({cms_az1_cidr})"
        )
        self.cms_sg.add_ingress_rule(
            peer=ec2.Peer.ipv4(cms_az2_cidr),
            connection=ec2.Port.tcp(cms_http_port),
            description=f"Permit Web Requests from AZ2 ({cms_az2_cidr})"
        )
        self.cms_sg.add_ingress_rule(
            peer=ec2.Peer.ipv4(bastion_cidr),
            connection=ec2.Port.tcp(cms_ssh_port),
            description=f"Permit SSH from Bastion ({bastion_cidr})"
        )
```

3. Configuración de las instancias EC2

Ahora, configuraremos la instancia del bastion host:

```
from aws_cdk import aws_ec2 as ec2
from aws_cdk import aws_iam as iam
from constructs import Construct


class BastionHost(Construct):
    def __init__(
        self,
        scope: Construct,
        id: str,
        vpc: ec2.IVpc,
        security_group: ec2.ISecurityGroup,
        **kwargs
    ):
        super().__init__(scope, id, **kwargs)

        instance_type = scope.node.try_get_context("bastionInstanceType") or "t2.micro"
        key_name = scope.node.try_get_context("bastionKeyName") or "default-key"
        role_name = scope.node.try_get_context("bastionRoleName") or "LabRole"
        volume_size = int(scope.node.try_get_context("bastionVolumeSize") or 8)
        volume_type = scope.node.try_get_context("bastionVolumeType") or "gp2"

        ami_ssm = scope.node.try_get_context("bastionAmiParameter") or \
            "/aws/service/canonical/ubuntu/server/22.04/stable/current/amd64/hvm/ebs-gp2/ami-id"

        ubuntu_ami = ec2.MachineImage.from_ssm_parameter(
            ami_ssm,
            os=ec2.OperatingSystemType.LINUX
        )

        key_pair = ec2.KeyPair.from_key_pair_name(
            self, "BastionKeyPair",
            key_pair_name=key_name
        )

        lab_role = iam.Role.from_role_name(self, "ImportedLabRole", role_name)

        public_subnet = vpc.public_subnets[0]

        self.instance = ec2.Instance(
            self, "Instance",
            instance_type=ec2.InstanceType(instance_type),
            machine_image=ubuntu_ami,
            vpc=vpc,
            vpc_subnets=ec2.SubnetSelection(subnets=[public_subnet]),
            security_group=security_group,
            role=lab_role,
            key_pair=key_pair,
            **kwargs
        )

        self.instance.instance.add_property_override(
            "BlockDeviceMappings", [
                {
                    "DeviceName": "/dev/sda1",
                    "Ebs": {
                        "VolumeSize": volume_size,
                        "VolumeType": volume_type
                    }
                }
            ]
        )
```

Este bloque define cómo se lanzarán las instancias EC2 del CMS, cómo se escalarán automáticamente y cómo se integra todo en un stack de CDK.

- **Launch Template:** define la AMI, tipo de instancia, key pair, y configuración inicial.

- **AutoScaling Group (ASG):** crea instancias basadas en el Launch Template y permite escalado automático.
- **Stack:** instancia la VPC, el ALB (Load Balancer) y el ASG. Es el punto de entrada del despliegue.

4. Configuración de la DB con RDS

Emplearemos el servicio RDS para la base de datos de nuestra infraestructura, por lo que crearemos el archivo rds.py y escribiremos lo siguiente en el:

```
from aws_cdk import (
    aws_ec2 as ec2,
    aws_rds as rds,
    aws_secretsmanager as secretsmanager,
    SecretValue,
    Duration,
    RemovalPolicy,
)
from constructs import Construct


class RdsDatabase(Construct):
    def __init__(
        self,
        scope: Construct,
        id: str,
        vpc: ec2.IVpc,
        security_group: ec2.ISecurityGroup,
        **kwargs
    ):
        super().__init__(scope, id, **kwargs)

        instance_class = scope.node.try_get_context("dbInstanceClass") or "t
3.micro"
        db_username = scope.node.try_get_context("dbMasterUser") or "wpu
ser"
        db_password = scope.node.try_get_context("dbMasterPassword") or
"wppassword"
```

```
        db_identifier = scope.node.try_get_context("dbIdentifier") or "drupal"
        allocated_storage = int(scope.node.try_get_context("dbAllocatedStorage") or 20)
        storage_type = rds.StorageType.GP2

        private_subnets = vpc.private_subnets
        if len(private_subnets) < 2:
            raise ValueError("El VPC debe tener al menos 2 subredes privadas para el DB Subnet Group.")

        subnet_selection = ec2.SubnetSelection(subnets=[private_subnets[0], private_subnets[1]])

        db_subnet_group = rds.SubnetGroup(
            self, "DbSubnetGroup",
            vpc=vpc,
            description="Subnet group for RDS",
            subnet_group_name="db-subnet-group",
            vpc_subnets=subnet_selection,
            removal_policy=RemovalPolicy.DESTROY
        )

        secret_arn = scope.node.try_get_context("dbSecretArn")
        if secret_arn:
            db_secret = secretsmanager.Secret.from_secret_arn(self, "ImportedDbSecret", secret_arn)
            credentials = rds.Credentials.from_secret(db_secret)
        else:
            db_secret = secretsmanager.Secret(
                self, "DbCredentialsSecret",
                secret_name=f"{db_identifier}-credentials",
                generate_secret_string=secretsmanager.SecretStringGenerator(
                    secret_string_template=f'{{"username":"{db_username}"}}',
                    generate_string_key="password",
                    exclude_punctuation=True,
                    password_length=16
                )
            )
```

```
    credentials = rds.Credentials.from_secret(db_secret)

engine = rds.DatabaseInstanceEngine.maria_db(
    version=rds.MariaDbEngineVersion.VER_10_5
)

try:
    instance_type = ec2.InstanceType(instance_class)
except Exception:
    instance_type = ec2.InstanceType("t3.micro")

db_instance = rds.DatabaseInstance(
    self, "DrupalRDS",
    engine=engine,
    credentials=credentials,
    instance_identifier=db_identifier,
    instance_type=instance_type,
    vpc=vpc,
    vpc_subnets=subnet_selection,
    security_groups=[security_group],
    multi_az=True,
    allocated_storage=allocated_storage,
    storage_type=storage_type,
    publicly_accessible=False,
    database_name="drupal",
    deletion_protection=False,
    backup_retention=Duration.days(0),
    enable_performance_insights=False,
    removal_policy=RemovalPolicy.DESTROY
)

db_instance.node.add_dependency(db_subnet_group)

self.db_instance = db_instance
self.db_secret = db_secret
self.db_subnet_group = db_subnet_group
```

5. AutoScaling y Stack principal

Definir el Launch Template, AutoScaling Group y Stack principal

Creamos un archivo llamado auto_scaling.py dentro de network:

```
from aws_cdk import (
    Duration,
    aws_autoscaling as autoscaling,
    aws_ec2 as ec2,
    aws_iam as iam
)
from constructs import Construct
from .security_groups import CmsSecurityGroups


class CmsAutoScaling(Construct):
    def __init__(
        self,
        scope: Construct,
        id: str,
        vpc: ec2.IVpc,
        **kwargs
    ):
        super().__init__(scope, id, **kwargs)

        # Parametrización desde context
        instance_type = scope.node.try_get_context("asgInstanceType") or "t2.micro"
        role_name = scope.node.try_get_context("asgRoleName") or "LabRole"
        min_capacity = int(scope.node.try_get_context("asgMinCapacity") or 2)
        max_capacity = int(scope.node.try_get_context("asgMaxCapacity") or 4)
        health_grace_minutes = int(scope.node.try_get_context("asgHealthGraceMinutes") or 3)
        # AMI parametrizada (default: la que tenías fija)
        ami_id = scope.node.try_get_context("asgAmiId") or "ami-0c02fb55956c7d316"

        security_groups = CmsSecurityGroups(self, "CmsSecurityGroups", vpc=vpc)

        custom_ami = ec2.MachineImage.generic_linux({
            'us-east-1': ami_id
        })

        lab_role = iam.Role.from_role_name(self, "ImportedAsgRole", role_name)

        # Crear Launch Template
        launch_template = ec2.LaunchTemplate(
            self, "CmsLaunchTemplate",
            instance_type=ec2.InstanceType(instance_type),
            machine_image=custom_ami,
            security_group=security_groups.cms_sg,
            role=lab_role,
            user_data=ec2.UserData.for_linux()
        )

        self.asg = autoscaling.AutoScalingGroup(
            self, "CmsASG",
            vpc=vpc,
            launch_template=launch_template,
            min_capacity=min_capacity,
            max_capacity=max_capacity,
            vpc_subnets=ec2.SubnetSelection(
                subnet_type=ec2.SubnetType.PRIVATE_WITH_EGRESS
            ),
            health_check=autoscaling.HealthCheck.elb(
                grace=Duration.minutes(health_grace_minutes)
            )
        )
```

Ahora, en el archivo cdk_taller_stack.py ponemos:

```
from aws_cdk import Stack, Duration, aws_ec2 as ec2
from aws_cdk import aws_iam as iam
from constructs import Construct
from aws_cdk import aws_elasticloadbalancingv2 as elbv2
from .network.vpc import CmsVpc
from .network.security_groups import BastionHostSG, DatabaseSG, LoadBa
lancerSG, CmsSecurityGroups
from .network.bastion_host import BastionHost
from .network.database import DatabaseInstance
from .network.auto_scaling import CmsAutoScaling

class CdkTallerStack(Stack):
    def __init__(self, scope: Construct, construct_id: str, **kwargs) → None:
        super().__init__(scope, construct_id, **kwargs)

        self.vpc = CmsVpc(self, "CMSVPC")

        self.bastion_sg = BastionHostSG(self, "BastionHostSG", vpc=self.vpc)
        self.db_sg = DatabaseSG(self, "DatabaseSG", vpc=self.vpc)
        self.lb_sg = LoadBalancerSG(self, "LoadBalancerSG", vpc=self.vpc)
        self.cms_sgs = CmsSecurityGroups(self, "CmsSecurityGroups", vpc=s
elf.vpc)

        self.bastion_host = BastionHost(
            self, "BastionHost",
            vpc=self.vpc,
            security_group=self.bastion_sg
        )

        self.database = DatabaseInstance(
            self, "DatabaseInstance",
            vpc=self.vpc,
            security_group=self.db_sg
        )
        cms_asg = CmsAutoScaling(
            self, "CmsAutoScaling",
            vpc=self.vpc
        )

        alb = elbv2.ApplicationLoadBalancer(
            self, "WebCMSALB",
            vpc=self.vpc,
            internet_facing=True,
            load_balancer_name="lb-WebCMS",
            security_group=self.lb_sg,
            vpc_subnets=ec2.SubnetSelection(
                subnet_type=ec2.SubnetType.PUBLIC
            )
        )

        listener = alb.add_listener(
            "HttpListener",
            port=80,
            open=True
        )

        listener.add_targets(
            "CmsTargets",
            port=80,
            targets=[cms_asg.asg],
            health_check={
                "path": "/",
                "interval": Duration.seconds(60),
                "healthy_threshold_count": 2,
                "unhealthy_threshold_count": 3
            }
        )
```

Sección 4: Despliegue

Para desplegar tuve algunos problemas. Primero, no es posible usar CDK deploy, la herramienta normal, debido a permisos de la cuenta de AWS Academy, así que la solución fue usar CDK synth, para obtener el archivo JSON de CloudFormation y subirlo a mano.

1. Generar el template de CloudFormation

Desde tu proyecto CDK, ejecutar:

```
cdk synth > cms_stack.template.json
```

Esto crea un archivo JSON llamado cms_stack.template.json con **toda la infraestructura definida en tu CDK**. Este archivo es compatible con AWS CloudFormation.

Despliegue manual en AWS

Debido a **restricciones de autorización en las cuentas de AWS Academy**, no fue posible usar cdk deploy. En su lugar, se subió manualmente el template de CloudFormation:

1. Entrar a la consola de AWS.
2. Ir a **CloudFormation → Create stack → With new resources (standard)**.
3. Seleccionar **Upload a template file** y subir cms_stack.json
4. Dar un nombre al stack y seguir el flujo de creación.

> Esto crea los mismos recursos definidos en CDK (VPC, ALB, AutoScaling Group) pero de forma manual.

---

5. Notas importantes

- El archivo JSON refleja toda la infraestructura definida en CDK, incluyendo **subredes, AMIs, Launch Templates y ALB**.
- Cualquier cambio en la configuración del CMS requiere **actualizar CDK y regenerar el JSON**.
- Aunque no se utilizó cdk deploy, este método permite **documentar y versionar la infraestructura** de manera reproducible.