Repaso de computación en la Nube y servicios AWS

Temas: VPC,Route53, CloudFront

Objetivos

Un desarrollador con conocimientos previos en VPC, Route 53 y CloudFront aprendería a replicar contenido en diferentes ubicaciones geográficas para mejorar la velocidad de entrega. Integrar CloudFront con otros servicios de AWS, como AWS SAM y AWS WAF, para mejorar la seguridad y el Crear registros de alias para enrutar el tráfico a CloudFront distribuciones, utilizar políticas de enrutamiento para dirigir el tráfico a diferentes servidores según sea necesario, asociar zonas alojadas de DNS privadas con varias VPC.

Los objetivos de este repaso son profundizar estos temas en tres niveles: desde el punto de vista teórico, dsde el punto de vista páctico usando las herramientas de AWS y por codificación a través de conceptos rudimentarios de python.

Tareas

Prepara una presentación grupal para responder las siguientes preguntas y utiliza un ide de tu preferencia para las implementaciones solicitadas:

Redes en AWS, VPC,Route53, CloudFront

1. Introduction to On-Premises Networks and Basic Corporate Networks

- **Pregunta:** Explica las diferencias clave entre una red on-premises y una red corporativa en la nube. Discute las ventajas y desventajas de cada enfoque desde la perspectiva de una empresa mediana.
- **Ejercicio:** Diseña un esquema de red básico para una empresa mediana que utiliza una red on-premises y otra que utiliza una red en la nube. Compara los costos, la seguridad y la escalabilidad de ambos diseños.

2. Fundamentals of IP Addressing and CIDRs

- **Pregunta:** Describe la estructura de una dirección IP IPv4 y una dirección IP IPv6. ¿Cuáles son las limitaciones de IPv4 que IPv6 intenta solucionar?
- **Ejercicio:** Calcula el rango de direcciones IP para la red 192.168.1.0/24. Luego, repite el cálculo para la red 192.168.1.0/26 y explica cómo cambia la cantidad de direcciones disponibles.

3. Network Sizes and Classes

- **Pregunta:** Explica la diferencia entre las clases de redes (A, B, C) y su uso histórico en la asignación de direcciones IP.
- **Ejercicio:** Dada una dirección IP 172.16.0.0, determina su clase y calcula el rango de direcciones IP posibles dentro de esta clase. Luego, aplica una máscara de subred para crear 4 subredes de tamaño igual y muestra los rangos de direcciones IP para cada subred.

4. Virtual Private Clouds (VPCs) y Subnets

- **Pregunta:** Define qué es una VPC y cómo se utiliza en AWS. ¿Cuáles son los componentes clave de una VPC?
- **Ejercicio:** Diseña una VPC con tres subredes: una pública, una privada y una subred de bases de datos. Incluye una puerta de enlace de Internet para la subred pública y una NAT Gateway para permitir que las instancias en la subred privada accedan a Internet. Explica cómo configurarías las tablas de rutas para permitir la comunicación adecuada entre estas subredes.

5. DNS y Global Routing with Amazon Route53

- **Pregunta:** ¿Qué es Amazon Route53 y cuáles son sus principales funciones? Explica cómo Route53 maneja la resolución de nombres y el enrutamiento de tráfico.
- **Ejercicio:** Configura un dominio en Amazon Route53, creando una zona alojada y agregando registros A y CNAME. Describe cómo configurar un balanceo de carga de tráfico utilizando políticas de enrutamiento basadas en geolocalización.

6. Implementando a Robust CDN con Amazon CloudFront

- **Pregunta:** Explica cómo Amazon CloudFront mejora la entrega de contenido. ¿Qué es una distribución de CloudFront y cuáles son sus componentes?
- **Ejercicio:** Diseña una configuración de CloudFront para una aplicación web global. Incluye la elección de una clase de precio y la configuración de políticas de caché. Explica cómo manejarías las actualizaciones de contenido y la invalidación de caché.

7. Introducción a Amazon API Gateway

- **Pregunta:** Describe las principales características de Amazon API Gateway. ¿Cómo facilita la gestión de APIs RESTful?
- **Ejercicio:** Diseña una API simple utilizando Amazon API Gateway que gestione las operaciones CRUD (Crear, Leer, Actualizar, Borrar) para una base de datos de usuarios. Explica cómo integrarías esta API con AWS Lambda para manejar las solicitudes.

Ejercicios prácticos en Python:

1. Automatización de la configuración de una VPC

- **Ejercicio:** Escribe un script en Python utilizando Boto3 que cree una VPC con dos subredes (una pública y una privada), configure una puerta de enlace de Internet y una NAT Gateway, y establezca las tablas de rutas necesarias.

2. Gestión de registros DNS con Route53

- **Ejercicio:** Utiliza Boto3 para escribir un script en Python que cree una zona alojada en Route53 y agregue registros A y CNAME. Incluye la configuración de un balanceo de carga de tráfico basado en latencia.

3. Configuración de una distribución de CloudFront

- **Ejercicio:** Escribe un script en Python que configure una distribución de CloudFront para un bucket de S3 que aloja contenido estático. Incluye la configuración de políticas de caché y la invalidación de objetos.

4. Implementación de una API con API Gateway y AWS Lambda

- **Ejercicio:** Desarrolla un script en Python que cree una API en API Gateway con endpoints para las operaciones CRUD. Utiliza AWS Lambda para escribir las funciones de backend que manejarán estas operaciones.

Preguntas de repaso

Diseño de redes corporativas:

Describe los componentes clave de una red corporativa básica y explique cómo cada componente contribuye a la funcionalidad y seguridad de la red.

Direcciones IP y CIDR:

Explica el proceso de asignación de direcciones IP utilizando CIDR y compare las ventajas y desventajas del uso de CIDR frente a las clases tradicionales de direcciones IP.

IPv4 vs IPv6:

Compara y contrasta las limitaciones de IPv4 con las características de IPv6. Discuta cómo IPv6 aborda los problemas de escalabilidad y seguridad presentes en IPv4.

Máscaras de subred y subneteo:

Define qué es una máscara de subred y describe su rol en el subneteo. Proporciona un ejemplo detallado de cómo realizar el subneteo en una red dada.

Routing y políticas de enrutamiento:

Explica la importancia de las políticas de enrutamiento en la gestión del tráfico de red. Describa al menos tres tipos de políticas de enrutamiento que se pueden implementar con Amazon Route53.

Conceptos de VPC:

Define qué es una VPC y explique cómo se diferencia de una red tradicional on-premises. Discuta las ventajas de usar VPCs en una arquitectura de nube.

Seguridad en VPCs:

Describe las mejores prácticas para asegurar una VPC. Incluya aspectos como listas de control de acceso (ACLs), grupos de seguridad y Network Address Translation (NAT).

NAT y VPC Peering:

Explica cómo funciona la traducción de direcciones de red (NAT) y cómo puede ser utilizada en una VPC. Además, discuta el concepto de VPC Peering y sus casos de uso.

VPNs y Direct Connect:

Compara las soluciones de VPNs con AWS Direct Connect. Explique los escenarios en los cuales se utilizaría una solución sobre la otra.

Amazon Route53:

Explica cómo funciona Amazon Route53 y sus componentes clave como hosted zones y health checks. Proporcione ejemplos de casos de uso para políticas de enrutamiento.

CDN con CloudFront:

Describe los beneficios de usar una CDN como Amazon CloudFront. Explique cómo se determina la clase de precios adecuada para una distribución CloudFront y los factores a considerar.

Amazon API Gateway:

Describe los componentes y funcionalidades principales de Amazon API Gateway. Explique cómo se puede utilizar para gestionar, asegurar y escalar APIs.

Usa el laboratorio de AWS Lab Learner

Ejercicio 1: Creación y gestión de usuarios IAM

- **Tarea:** Crea tres usuarios IAM con diferentes niveles de permisos. Asigna políticas específicas a cada uno y documenta los pasos realizados.

Ejercicio 2: Configuración de MFA para usuarios IAM

- **Tarea:** Configura MFA para un usuario IAM y documenta el proceso, incluyendo cómo verificar el estado de MFA y cómo manejar la autenticación de doble factor.

Ejercicio 3: Definición y asignación de políticas IAM

- **Tarea:** Crea una política personalizada que otorgue permisos específicos a un bucket de S3. Asigna esta política a un usuario o grupo y verifica que los permisos funcionen correctamente.

Ejercicio 4: Implementación de Roles IAM para credenciales temporales

- **Tarea:** Configura un rol IAM que permita a una instancia EC2 acceder a un bucket de S3. Documenta el proceso y prueba el acceso desde la instancia.

Ejercicio 5: Configuración de una VPC con dubredes y NAT Gateway

Objetivo: Crear y configurar una VPC con subredes públicas y privadas, y configurar una NAT Gateway para permitir que las instancias en las subredes privadas accedan a Internet.

Pasos:

Crear una VPC:

- Utiliza el AWS Management Console para crear una nueva VPC con un rango CIDR de 10.0.0.0/16.

Crear Subredes:

- Crea una subred pública con el rango 10.0.1.0/24 en una zona de disponibilidad (AZ).
- Crea una subred privada con el rango 10.0.2.0/24 en la misma AZ.

Configurar la puerta de enlace de internet:

- Adjunta una Internet Gateway a la VPC.
- Actualiza la tabla de rutas de la subred pública para que apunte a la Internet Gateway.

Configurar una NAT Gateway:

- Crea una NAT Gateway en la subred pública.
- Actualiza la tabla de rutas de la subred privada para que apunte a la NAT Gateway.

Lanzar Instancias EC2:

- Lanza una instancia EC2 en la subred pública y otra en la subred privada.
- Configura las instancias para que puedan comunicarse entre sí y prueben la conectividad a Internet desde la subred privada utilizando la NAT Gateway.

Validación:

- Asegúrate de que la instancia en la subred privada pueda acceder a Internet.
- Comprueba que ambas instancias puedan comunicarse entre sí.

Ejercicio 2: Configuración de DNS con Amazon Route53

Objetivo: Configurar un dominio en Route53, crear registros A y CNAME, y configurar un balanceo de carga basado en geolocalización.

Pasos:

Configurar una Zona Alojada:

- En Route53, crea una nueva zona alojada para un dominio que poseas o un subdominio de example.com.

Agregar registros DNS:

- Crea un registro A para apuntar a la dirección IP de una instancia EC2.
- Crea un registro CNAME para apuntar a un nombre de dominio alternativo.

Configurar balanceo de carga:

- Utiliza políticas de enrutamiento basadas en geolocalización para distribuir el tráfico entre dos instancias EC2 en diferentes regiones.

Probar la configuración:

- Utiliza herramientas como dig o nslookup para verificar que los registros DNS se resuelven correctamente.
- Accede a las instancias EC2 utilizando los registros DNS y verifica que el balanceo de carga basado en geolocalización funciona según lo esperado.

Validación:

- Verifica que los registros DNS se resuelvan correctamente.
- Asegúrate de que el balanceo de carga dirija el tráfico a la instancia correcta según la geolocalización del cliente.

Ejercicio 3: Implementación de una CDN con Amazon CloudFront

Objetivo: Configurar una distribución de CloudFront para entregar contenido estático desde un bucket de S3, incluyendo la configuración de políticas de caché y la invalidación de objetos.

Pasos:

Configurar un Bucket de S3:

- Crea un bucket de S3 y carga algunos archivos estáticos (por ejemplo, imágenes, HTML, CSS).

Crear una distribución de CloudFront:

- En el AWS Management Console, crea una nueva distribución de CloudFront que apunte al bucket de S3.
- Configura políticas de caché para controlar la duración del almacenamiento en caché de los objetos.

Configurar políticas de precio:

- Selecciona una clase de precio adecuada para la distribución de CloudFront basada en la cobertura geográfica de tus usuarios.

Invalidar objetos:

- Utiliza la consola de CloudFront para invalidar objetos específicos y forzar su actualización en la caché.

Validación:

- Accede al contenido a través de la URL de CloudFront y verifica que se entrega correctamente desde el edge location más cercano.
- Verifica que las políticas de caché se apliquen correctamente y que la invalidación de objetos funcione como se espera.

Ejercicio 4: Implementación de una API con Amazon API Gateway y AWS Lambda

Objetivo: Crear una API RESTful con Amazon API Gateway y utilizar AWS Lambda para manejar las solicitudes.

Pasos:

Configurar una API en API Gateway:

- En el AWS Management Console, crea una nueva API RESTful en API Gateway.

Crear Endpoints para Operaciones CRUD:

- Define endpoints para las operaciones Crear, Leer, Actualizar y Borrar.
- Configura métodos HTTP (POST, GET, PUT, DELETE) para cada endpoint.

Integrar AWS Lambda:

- Crea funciones Lambda para manejar cada operación CRUD.
- Configura API Gateway para que invoque las funciones Lambda correspondientes para cada endpoint.

Probar la API:

- Utiliza herramientas como Postman o curl para enviar solicitudes a la API y verificar que las operaciones CRUD funcionan correctamente.

Validación:

- Verifica que las funciones Lambda se invoquen correctamente desde API Gateway.
- Asegúrate de que las operaciones CRUD se realicen con éxito en una base de datos de prueba.

Código

1. Introducción a las Redes On-Premises y Corporativas Básicas

Simula la topología de una red y la comunicación entre dispositivos utilizando Python y librerías como networkx para modelar y visualizar las redes.

```
import networkx as nx

import matplotlib.pyplot as plt


# Crear un grafo vacío

G = nx.Graph()
```

```
# Añadir nodos (dispositivos)

G.add_node("Router")

G.add_node("Switch 1")

G.add_node("Switch 2")

G.add_node("PC 1")

G.add_node("PC 2")

G.add_node("PC 3")

G.add_node("PC 4")


# Añadir enlaces (conexiones)

G.add_edges_from([("Router", "Switch 1"), ("Router", "Switch 2"),

                  ("Switch 1", "PC 1"), ("Switch 1", "PC 2"),

                  ("Switch 2", "PC 3"), ("Switch 2", "PC 4")])


# Dibujar el grafo

nx.draw(G, with_labels=True, node_size=3000, node_color='skyblue', font_size=10,
font_color='black')

plt.show()
```

2. Direccionamiento IP y CIDR

Podemos escribir un script en Python que calcule y muestre rangos de direcciones IP basados en CIDR.

```
from ipaddress import ip_network


# Rango CIDR

cidr = '192.168.1.0/24'
```


```
# Crear una red basada en el CIDR

network = ip_network(cidr)


# Mostrar todas las direcciones IP en la red

print(f"Rango de direcciones IP para {cidr}:")

for ip in network.hosts():

    print(ip)
```

3. Redes privadas virtuales (VPCs) y subredes

Podemos simular la creación de VPCs y subredes usando clases en Python para representar estos elementos.

```
class VPC:

    def __init__(self, cidr):

        self.cidr = cidr

        self.subnets = []


    def add_subnet(self, cidr):

        subnet = Subnet(cidr)

        self.subnets.append(subnet)


class Subnet:

    def __init__(self, cidr):

        self.cidr = cidr


# Crear una VPC
```

```
vpc = VPC('10.0.0.0/16')


# Añadir subredes
vpc.add_subnet('10.0.1.0/24')
vpc.add_subnet('10.0.2.0/24')


# Mostrar la configuración de la VPC
print(f"VPC CIDR: {vpc.cidr}")
for subnet in vpc.subnets:
    print(f"Subred CIDR: {subnet.cidr}")
```

4. DNS y enrutamiento global con simulación básica de Route53

Podemos simular la resolución de nombres y el enrutamiento básico utilizando diccionarios en Python.

```
# Simulación de un DNS con registros A y CNAME
dns_records = {
    'example.com': '192.168.1.10',
    'www.example.com': 'example.com',
}
def resolve_dns(name):
    if name in dns_records:
        value = dns_records[name]
        # Resolver CNAME
        if value in dns_records:
            value = dns_records[value]
        return value
```

```
    return None


# Probar la resolución de DNS

print(f"IP de example.com: {resolve_dns('example.com')}")

print(f"IP de www.example.com: {resolve_dns('www.example.com')}")
```

5. Implementación de una CDN básica con Python

Podemos simular una CDN que almacena en caché el contenido y responde a las solicitudes.

```
import time

class CDN:
    def __init__(self):
        self.cache = {}

    def get_content(self, url):
        if url in self.cache:
            print("Content served from cache")
            return self.cache[url]
        else:
            content = self.fetch_from_origin(url)
            self.cache[url] = content
            return content

    def fetch_from_origin(self, url):
        print("Fetching content from origin server...")
        time.sleep(2)  # Simular tiempo de respuesta del servidor de origen
        return f"Content of {url}"

# Crear una instancia de CDN
cdn = CDN()

# Solicitar contenido
print(cdn.get_content("https://example.com/image.png"))
print(cdn.get_content("https://example.com/image.png"))
```

Ejercicio 6: Simulación Completa de una Red Corporativa

Objetivo: Diseñar y simular una red corporativa completa utilizando Python. La red debe incluir routers, switches, y dispositivos finales con direccionamiento IP. Se debe implementar el enrutamiento estático y dinámico entre subredes, y mostrar la topología de la red.

Instrucciones:

Diseñar la red:

- Crea clases para representar routers, switches y dispositivos finales (PCs, servidores, etc.).
- Establece conexiones entre estos dispositivos para formar la red.

Direccionamiento IP:

- Asigna direcciones IP a cada dispositivo según las subredes.
- Implementa una función para calcular y asignar subredes utilizando CIDR.

Enrutamiento:

- Implementa enrutamiento estático entre subredes.
- Simula el protocolo de enrutamiento OSPF para enrutamiento dinámico.

Visualización:

- Muestra la topología de la red utilizando networkx y matplotlib.

Simulación de tráfico:

- Implementar funciones para enviar paquetes de datos entre dispositivos y mostrar el camino que sigue el paquete a través de la red.

Código base:

```
import networkx as nx

import matplotlib.pyplot as plt

from ipaddress import ip_network, ip_address


class Device:

    def __init__(self, name, device_type):

        self.name = name
```


```
        self.device_type = device_type
        self.ip_address = None

class Router(Device):
    def __init__(self, name):
        super().__init__(name, 'Router')
        self.routing_table = {}

    def add_route(self, network, next_hop):
        self.routing_table[network] = next_hop

    def get_route(self, ip):
        for network, next_hop in self.routing_table.items():
            if ip in ip_network(network):
                return next_hop
        return None
class Switch(Device):
    def __init__(self, name):
        super().__init__(name, 'Switch')
class PC(Device):
    def __init__(self, name):
        super().__init__(name, 'PC')
class Network:
    def __init__(self):
        self.devices = []
```

```
        self.edges = []

    def add_device(self, device):
        self.devices.append(device)

    def connect(self, device1, device2):
        self.edges.append((device1, device2))

    def assign_ip_addresses(self, cidr):
        network = ip_network(cidr)
        hosts = network.hosts()
        for device in self.devices:
            if isinstance(device, PC):
                device.ip_address = str(next(hosts))

    def visualize(self):
        G = nx.Graph()
        for device in self.devices:
            G.add_node(device.name)
        for edge in self.edges:
            G.add_edge(edge[0].name, edge[1].name)
        nx.draw(G, with_labels=True, node_size=3000, node_color='skyblue', font_size=10,
font_color='black')
        plt.show()
```


```
    def simulate_traffic(self, src, dst):
        path = nx.shortest_path(self._build_graph(), source=src.name, target=dst.name)
        print(f"Path from {src.name} to {dst.name}: {' -> '.join(path)}")

    def _build_graph(self):
        G = nx.Graph()
        for device in self.devices:
            G.add_node(device.name)
        for edge in self.edges:
            G.add_edge(edge[0].name, edge[1].name)
        return G

# Crear la red y dispositivos
network = Network()
router1 = Router("Router1")
router2 = Router("Router2")
switch1 = Switch("Switch1")
switch2 = Switch("Switch2")
pc1 = PC("PC1")
pc2 = PC("PC2")
pc3 = PC("PC3")
pc4 = PC("PC4")

# Añadir dispositivos a la red
network.add_device(router1)
```


```
network.add_device(router2)

network.add_device(switch1)

network.add_device(switch2)

network.add_device(pc1)

network.add_device(pc2)

network.add_device(pc3)

network.add_device(pc4)


# Conectar dispositivos

network.connect(router1, switch1)

network.connect(router1, switch2)

network.connect(switch1, pc1)

network.connect(switch1, pc2)

network.connect(switch2, pc3)

network.connect(switch2, pc4)

network.connect(router1, router2)


# Asignar direcciones IP

network.assign_ip_addresses('192.168.1.0/24')


# Añadir rutas estáticas

router1.add_route('192.168.1.0/24', 'Router1')

router1.add_route('192.168.2.0/24', 'Router2')

router2.add_route('192.168.1.0/24', 'Router1')

router2.add_route('192.168.2.0/24', 'Router2')
```


```
# Visualizar la red

network.visualize()


# Simular tráfico

network.simulate_traffic(pc1, pc3)
```

Ejercicio 7: Implementación de un Servicio de DNS Dinámico

Objetivo: Implementa un servicio de DNS dinámico en Python que pueda gestionar múltiples registros, incluyendo registros A, CNAME y MX, y simular resoluciones de nombres de dominio.

Instrucciones:

Define la clase DNS:

- Crea una clase DNS para gestionar registros DNS.
- Implementa métodos para agregar, eliminar y actualizar registros.

Simular resolución DNS:

- Implementa un método para resolver nombres de dominio a direcciones IP.
- Soporta registros A, CNAME y MX.

Interfaz de usuario:

- Implementa una interfaz de línea de comandos (CLI) para interactuar con el servicio DNS.
- Permite a los usuarios agregar, eliminar, actualizar y resolver registros DNS.

Código base:

```
class DNS:

    def __init__(self):

        self.records = {

            'A': {},

            'CNAME': {},
```


```
        'MX': {}
    }

    def add_record(self, record_type, name, value):
        if record_type in self.records:
            self.records[record_type][name] = value
        else:
            print(f"Record type {record_type} not supported")

    def delete_record(self, record_type, name):
        if record_type in self.records and name in self.records[record_type]:
            del self.records[record_type][name]
        else:
            print(f"Record {name} not found in {record_type} records")

    def update_record(self, record_type, name, value):
        if record_type in self.records and name in self.records[record_type]:
            self.records[record_type][name] = value
        else:
            print(f"Record {name} not found in {record_type} records")

    def resolve(self, name):
        if name in self.records['A']:
            return self.records['A'][name]
        elif name in self.records['CNAME']:
```


```
            cname = self.records['CNAME'][name]
            return self.resolve(cname)
        elif name in self.records['MX']:
            return self.records['MX'][name]
        else:
            return None

    def __str__(self):
        return str(self.records)

# Simulación de la CLI
def main():
    dns = DNS()
    while True:
        command = input("Enter command (add, delete, update, resolve, exit): ")
        if command == 'add':
            record_type = input("Enter record type (A, CNAME, MX): ")
            name = input("Enter name: ")
            value = input("Enter value: ")
            dns.add_record(record_type, name, value)
        elif command == 'delete':
            record_type = input("Enter record type (A, CNAME, MX): ")
            name = input("Enter name: ")
            dns.delete_record(record_type, name)
        elif command == 'update':
```

```
            record_type = input("Enter record type (A, CNAME, MX): ")
            name = input("Enter name: ")
            value = input("Enter new value: ")
            dns.update_record(record_type, name, value)
        elif command == 'resolve':
            name = input("Enter name to resolve: ")
            ip = dns.resolve(name)
            if ip:
                print(f"IP for {name} is {ip}")
            else:
                print(f"{name} not found")
        elif command == 'exit':
            break
        else:
            print("Invalid command")

if __name__ == "__main__":
    main()
```

Ejercicio 8: Simulación de un CDN con control de caché avanzado

Objetivo: Implementar una CDN que almacene en caché contenido y soporte políticas de caché avanzadas como expiración basada en tiempo y tamaño máximo de caché.

Instrucciones:

Define la clase CDN:

- Crea una clase CDN para gestionar el contenido en caché.
- Implementa métodos para agregar contenido a la caché, recuperar contenido, y manejar la expiración de caché.

Políticas de caché:

- Implementa una política de caché basada en tiempo para expirar contenido después de un tiempo determinado.
- Implementa una política de tamaño máximo de caché y manejar la eliminación de contenido cuando se exceda el tamaño.

Simular solicitudes de contenido:

- Implementa un método para simular solicitudes de contenido que respete las políticas de caché.

Código base:

```
import time


class CDN:
    def __init__(self, max_cache_size, cache_expiration):
        self.cache = {}
        self.max_cache_size = max_cache_size
        self.cache_expiration = cache_expiration

    def get_content(self, url):
        if url in self.cache:
            content, timestamp = self.cache[url]
            if time.time() - timestamp < self.cache_expiration:
                print("Content served from cache")
                return content
            else:
                print("Cache expired, fetching new content")
                self.cache.pop(url)
```

```
        content = self.fetch_from_origin(url)
        self.add_to_cache(url, content)
        return content

    def fetch_from_origin(self, url):
        print("Fetching content from origin server...")
        time.sleep(2)  # Simular tiempo de respuesta del servidor de origen
        return f"Content of {url}"

    def add_to_cache(self, url, content):
        if len(self.cache) >= self.max_cache_size:
            self.evict_cache()
        self.cache[url] = (content, time.time())

    def evict_cache(self):
        # Evict the oldest cache entry
        oldest_url = min(self.cache, key=lambda k: self.cache[k][1])
        print(f"Evicting cache for {oldest_url}")
        self.cache.pop(oldest_url)

    def __str__(self):
        return str(self.cache)

# Simulación de la CDN
def main():
```


```
    cdn = CDN(max_cache_size=3, cache_expiration=10)

    while True:

        command = input("Enter command (get, exit): ")

        if command == 'get':

            url = input("Enter URL to fetch: ")

            content = cdn.get_content(url)

            print(content)

        elif command == 'exit':

            break

        else:

            print("Invalid command")


if __name__ == "__main__":

    main()
```

Ejercicio 9: Implementación de una API RESTful con Python

Objetivo: Implementar una API RESTful que gestione recursos utilizando un framework web ligero como Flask. La API debe soportar operaciones CRUD y almacenar datos en una estructura de datos en memoria.

Instrucciones:

Define la API:

- Utiliza Flask para definir endpoints para las operaciones CRUD.
- Implementa métodos para crear, leer, actualizar y eliminar recursos.

Estructura de datos:

- Utiliza una estructura de datos en memoria (como un diccionario) para almacenar los recursos.

Simular solicitudes:

- Implementar una interfaz de línea de comandos (CLI) para enviar solicitudes a la API y mostrar los resultados.

Código base:

```
from flask import Flask, request, jsonify

app = Flask(__name__)

# Estructura de datos en memoria
resources = {}

@app.route('/resources', methods=['POST'])
def create_resource():
    resource_id = request.json['id']
    resource_data = request.json['data']
    resources[resource_id] = resource_data
    return jsonify({"message": "Resource created"}), 201

@app.route('/resources/<resource_id>', methods=['GET'])
def get_resource(resource_id):
    resource_data = resources.get(resource_id)
    if resource_data:
        return jsonify({"id": resource_id, "data": resource_data})
    return jsonify({"message": "Resource not found"}), 404
```

```
@app.route('/resources/<resource_id>', methods=['PUT'])
def update_resource(resource_id):
    resource_data = request.json['data']
    if resource_id in resources:
        resources[resource_id] = resource_data
        return jsonify({"message": "Resource updated"})
    return jsonify({"message": "Resource not found"}), 404

@app.route('/resources/<resource_id>', methods=['DELETE'])
def delete_resource(resource_id):
    if resource_id in resources:
        del resources[resource_id]
        return jsonify({"message": "Resource deleted"})
    return jsonify({"message": "Resource not found"}), 404

if __name__ == '__main__':
    app.run(debug=True)
```

Ejercicio 10: Simulación completa de una red corporativa con enrutamiento dinámico

Objetivo: Diseñar y simular una red corporativa completa utilizando Python. La red debe incluir routers, switches, y dispositivos finales con direccionamiento IP. Se debe implementar el enrutamiento dinámico entre subredes utilizando el protocolo OSPF.

Instrucciones:

Diseñar la red:

- Crea clases para representar routers, switches y dispositivos finales (PCs, servidores, etc.).
- Establece conexiones entre estos dispositivos para formar la red.

Direccionamiento IP:

- Asignar direcciones IP a cada dispositivo según las subredes.
- Implementa una función para calcular y asignar subredes utilizando CIDR.

Enrutamiento dinámico:

- Implementa el protocolo OSPF para enrutamiento dinámico.
- Simula la actualización de tablas de enrutamiento y la propagación de rutas.

Visualización:

- Muestra la topología de la red utilizando networkx y matplotlib.

Simulación de tráfico:

- Implementa funciones para enviar paquetes de datos entre dispositivos y mostrar el camino que sigue el paquete a través de la red.