**InterIntel Implementation Engineer Assessment – Paul Ndambo**

**1. Give examples of different integration protocols you have come across and give**

**example scripts in python3 on how to achieve each one. (10 pts)**

i. **RESTful API Integration**

Representational State Transfer is a widely used architectural style for designing networked applications. Python has several libraries for integrating REST APIs, requests is the most popular one.

import requests

# Send a GET request to the API endpoint

response = requests.get('https://jsonplaceholder.typicode.com/posts/1')

# Check the status code of the response

if response.status\_code == 200:

# Parsing the JSON response

data = response.json()

print(data)

else:

print('Error:', response.status\_code)

ii. **SOAP API Integration**

Simple Access Object Protocol is an XML-based protocol for exchanging structured information on web services. Python's zeep library is commonly used for SOAP integrations.

from zeep import Client

# Define the SOAP service URL

soap\_url = 'http://webservices.oorsprong.org/websamples.countryinfo/CountryInfoService.wso?WSDL'

# Create a SOAP client

client = Client(soap\_url)

# Make a SOAP request to get information about a specific country (e.g., United States)

response = client.service.FullCountryInfo('USA')

# Print the response

print(response)

**iii. AMQP**

Advanced Message Queuing Protocol is a message-oriented middleware protocol for asynchronous communication between applications.

## Publisher

import os

import pika

import os

import pika

import json

AMQP\_URL = 'amqps://ncvxnkvq:Vpf8yc\_tTiFuqfXj\_7hDqaiDGH4LYmCy@hummingbird.rmq.cloudamqp.com/ncvxnkvq'

class BasePublisher(object):

def \_\_init\_\_(self, routing\_key, body) -> None:

self.exchange = "test\_exchange"

self.queue = "test\_queue"

self.routing\_key = routing\_key

self.url = AMQP\_URL

self.body = body

def run(self):

self.\_start\_connection()

self.\_publish\_message()

def \_start\_connection(self):

params = pika.URLParameters(self.url)

connection = pika.BlockingConnection(params)

channel = connection.channel()

channel.exchange\_declare(self.exchange)

channel.queue\_declare(queue=self.queue, durable=True)

channel.queue\_bind(self.queue, self.exchange, self.routing\_key)

return channel, connection

def \_publish\_message(self):

channel, connection = self.\_start\_connection()

channel.basic\_publish(

body=json.dumps(self.body),

exchange=self.exchange,

routing\_key=self.routing\_key

)

channel.close()

connection.close()

publisher = BasePublisher(

routing\_key="testing\_queue",

body={

"name": "Paul Ndambo",

"school": "Masinde Muliro University of Sci. & Tech.",

"programme": "Bsc. Information Technology",

"specialty": "Backend Software Engineering"

}

)

publisher.run()

## Consumer

import pika

class BaseConsumer:

def \_\_init\_\_(self, url, exchange\_name, queue\_name, routing\_key):

self.url = url

self.exchange\_name = exchange\_name

self.queue\_name = queue\_name

self.routing\_key = routing\_key

self.connection = None

self.channel = None

def \_start\_connection(self):

params = pika.URLParameters(self.url)

self.connection = pika.BlockingConnection(params)

self.channel = self.connection.channel()

# Declare a topic exchange

self.channel.exchange\_declare(exchange=self.exchange\_name, exchange\_type='topic')

# Declare a queue with a specific routing key binding

self.channel.queue\_declare(queue=self.queue\_name, durable=True)

self.channel.queue\_bind(exchange=self.exchange\_name, queue=self.queue\_name, routing\_key=self.routing\_key)

def callback(self, ch, method, properties, body):

print(f' [x] Received {body} with routing key: {method.routing\_key}')

def \_consume\_messages(self):

self.channel.basic\_consume(queue=self.queue\_name, on\_message\_callback=self.callback, auto\_ack=True)

print(f' [\*] Waiting for messages with routing key {self.routing\_key}:')

self.channel.start\_consuming()

def \_close\_connection(self):

if self.connection and self.connection.is\_open:

self.connection.close()

if \_\_name\_\_ == "\_\_main\_\_":

url = 'amqps://ncvxnkvq:Vpf8yc\_tTiFuqfXj\_7hDqaiDGH4LYmCy@hummingbird.rmq.cloudamqp.com/ncvxnkvq'

exchange\_name = 'topic\_exchange'

queue\_name = 'test\_queue'

routing\_key = 'test.\*'

consumer = BaseConsumer(url, exchange\_name, queue\_name, routing\_key)

consumer.\_start\_connection()

try:

consumer.\_consume\_messages()

except KeyboardInterrupt:

print('Interrupted. Closing connection...')

finally:

consumer.\_close\_connection()

**2. Give a walkthrough of how you will manage a data streaming application sending**

**one million notifications every hour while giving examples of technologies and**

**configurations you will use to manage load and asynchronous services. (10 pts)**

Managing a data streaming application that sends one million notifications every hour involves handling significant loads and implementing asynchronous services to ensure efficiency and scalability. Here's a walkthrough with examples of technologies and configurations:

**1. Load Management**

* Technology: **Kubernetes**
* Configuration: Deploy the notification generation application as a Kubernetes deployment with multiple replicas. This will automatically scale the number of replicas up or down based on the incoming load. To enable auto scaling, i will add an horizontal pod autoscaler (HPA)
* Technology: **Nginx**
* Configuration: Configure Nginx as a load balancer to distribute traffic evenly across the replicas of the notification generation application. In the case which is more kubernetes centered a nginx-ingress will do the job.
* Technology: **Prometheus**
* Configuration: Deploy Prometheus to monitor the load on the application and trigger alerts if the CPU or memory usage exceeds certain thresholds.

2. **Asynchronous Services**

* Technology: **RabbitMQ**, (Apache Kafka & AWS SQS would also make a good choice)
* Configuration: Deploy RabbitMQ as a distributed messaging system to store and deliver notifications asynchronously.
* Technology: **Celery**
* Configuration: Use Celery to consume messages from Apache and process notifications in a background manner.
* Technology: **Amazon Simple Notification Service (SNS)**
* Configuration: Use Amazon SNS to send notifications to email addresses or mobile devices.

3**.Fault Tolerance**

* Technology: **Docker**
* Configuration: Deploy the application using Docker containers to ensure that the environment is consistent across all replicas.
* Technology: **Redis**
* Configuration: Use Redis as a cache to store frequently accessed data and reduce the load on the database.

3**. Give examples of different encryption/hashing methods you have come across**

**(one-way and two-way) and give example scripts in python3 on how to achieve**

**each one. (20 pts)**

1. **One-way Hashing**
2. MD5 (Message Digest Algorithm 5)

import hashlib

def generate\_hash\_md5(data):

hashed = hashlib.md5(data.encode()).hexdigest()

return hashed

data = "Hello, World!"

hashed\_md5 = generate\_hash\_md5(data)

print(f"MD5 Hash: {hashed\_md5}")

1. SHA-256 (Secure Hash Algorithm 256-bit)

import hashlib

def generate\_hash\_sha256(data):

hashed = hashlib.sha256(data.encode()).hexdigest()

return hashed

data = "Hello, World!"

hashed\_sha256 = generate\_hash\_sha256(data)

print(f"SHA-256 Hash: {hashed\_sha256}")

1. **Two-way Hashing**

I. AES (Advanced Encryption Standard)

from base64 import b64decode, b64encode

from Crypto.Cipher import AES

from Crypto.Random import get\_random\_bytes

def encrypt\_aes(data, key):

cipher = AES.new(key, AES.MODE\_EAX)

ciphertext, tag = cipher.encrypt\_and\_digest(data.encode())

return b64encode(cipher.nonce + tag + ciphertext).decode()

def decrypt\_aes(encrypted\_data, key):

encrypted\_data = b64decode(encrypted\_data.encode())

nonce = encrypted\_data[:16]

tag = encrypted\_data[16:32]

ciphertext = encrypted\_data[32:]

cipher = AES.new(key, AES.MODE\_EAX, nonce=nonce)

decrypted\_data = cipher.decrypt\_and\_verify(ciphertext, tag)

return decrypted\_data.decode()

key = get\_random\_bytes(16)

data = "Sensitive information"

encrypted\_data = encrypt\_aes(data, key)

print(f"Encrypted Data: {encrypted\_data}")

decrypted\_data = decrypt\_aes(encrypted\_data, key)

print(f"Decrypted Data: {decrypted\_data}")

Ii. RSA (Rivest–Shamir–Adleman)

from Crypto.Cipher import PKCS1\_OAEP

from Crypto.PublicKey import RSA

## Commands to generate .pem file

# openssl genpkey -algorithm RSA -out private.pem

# openssl rsa -pubout -in private.pem -out public.pem

def encrypt\_rsa(data, public\_key\_path):

key = RSA.import\_key(open(public\_key\_path).read())

cipher = PKCS1\_OAEP.new(key)

ciphertext = cipher.encrypt(data.encode())

return ciphertext

def decrypt\_rsa(encrypted\_data, private\_key\_path):

key = RSA.import\_key(open(private\_key\_path).read())

cipher = PKCS1\_OAEP.new(key)

decrypted\_data = cipher.decrypt(encrypted\_data)

return decrypted\_data.decode()

public\_key\_path = "public.pem"

private\_key\_path = "private.pem"

data = "Secure communication"

encrypted\_data = encrypt\_rsa(data, public\_key\_path)

print(f"Encrypted Data: {encrypted\_data}")

decrypted\_data = decrypt\_rsa(encrypted\_data, private\_key\_path)

print(f"Decrypted Data: {decrypted\_data}")