from \_\_future\_\_ import print\_function

import boto3

import os

dynamodb\_resource = boto3.resource('dynamodb')

table = dynamodb\_resource.Table(os.environ["TABLE\_NAME"])

def handle(event, context):

for record in event['Records']:

if not record["dynamodb"].get("NewImage"):

print("Continuing cause no NewImage")

continue

if 'trafficJamIndicator' in record["dynamodb"]["NewImage"]:

unique\_id = record["dynamodb"]["NewImage"]["uniqueId"]['S']

alert\_indicator = record["dynamodb"]["NewImage"]["trafficJamIndicator"]['N']

location = record["dynamodb"]["NewImage"]["loc"]['S']

save\_traffic\_jam\_alert\_item(unique\_id, alert\_indicator, location)

def save\_traffic\_jam\_alert\_item(unique\_id, alert\_indicator, location):

try:

table.update\_item(

Key={

'uniqueId': unique\_id

},

UpdateExpression="set alertIndicator = :ai, loc=:l",

ExpressionAttributeValues={

':ai': alert\_indicator,

':l': location

},

ConditionExpression='alertIndicator <> :ai')

except dynamodb\_resource.meta.client.exceptions.ConditionalCheckFailedException as e:

None

import json

import logging

import os

from urllib.request import Request, urlopen

from urllib.error import URLError, HTTPError

SLACK\_CHANNEL = os.environ['SLACK\_CHANNEL']

SLACK\_WEBHOOK = os.environ['SLACK\_WEBHOOK']

FLASK\_API = os.environ['FLASK\_API']

logger = logging.getLogger()

logger.setLevel(logging.INFO)

def handle(event, context):

for record in event['Records']:

if not record["dynamodb"].get("NewImage"):

continue;

unique\_id = record["dynamodb"]["NewImage"]["uniqueId"]['S']

location = record["dynamodb"]["NewImage"]["loc"]['S']

alert\_indicator = record["dynamodb"]["NewImage"]["alertIndicator"]['S']

if alert\_indicator == "1":

message = f"\*NEW\* \*traffic jam\* at location: \*{location}\* :sob:"

api\_message = f"New traffic jam at location: {location}"

send\_message(message)

send\_api\_message(api\_message, True)

elif alert\_indicator == "0":

if not record["dynamodb"].get("OldImage"):

continue

else:

message = f"The \*traffic jam\* at location: \*{location}\* is \*GONE\*! :smile:"

api\_message = f"The traffic jam at location: {location} is gone!"

send\_message(message)

send\_api\_message(api\_message, False)

def send\_message(message):

slack\_message = {

'channel': SLACK\_CHANNEL,

'text': message

}

req = Request(SLACK\_WEBHOOK, json.dumps(slack\_message).encode('utf-8'))

try:

response = urlopen(req)

response.read()

logger.info("Message posted to %s", slack\_message['channel'])

except HTTPError as e:

logger.error("Request failed: %d %s", e.code, e.reason)

except URLError as e:

logger.error("Server connection failed: %s", e.reason)

def send\_api\_message(message, incoming):

api\_notification = {

'message': message,

'incoming': incoming

}

req\_api = Request(FLASK\_API, json.dumps(api\_notification).encode('utf-8'))

try:

response = urlopen(req\_api)

response.read()

logger.info("Message posted to flask api (for frontend integration)")

except HTTPError as e:

logger.error("Request failed: %d %s", e.code, e.reason)

except URLError as e:

logger.error("Server connection failed: %s", e.reason)

import boto3

import botocore

import base64

import os

import json

from decimal import Decimal

dynamodb\_resource = boto3.resource('dynamodb')

table = dynamodb\_resource.Table(os.environ["TABLE\_NAME"])

def handle(event, context):

dynamodb\_items = scan\_table()

data = [parse\_dynamo\_item\_to\_dict(item) for item in dynamodb\_items]

print(data)

response = {

"statusCode": 200,

"body": json.dumps(data)

}

return response

def scan\_table(input={}):

response = table.scan(\*\*input)

data = response['Items']

while 'LastEvaluatedKey' in response:

response = table.scan(ExclusiveStartKey=response['LastEvaluatedKey'], \*\*input)

data.extend(response['Items'])

return data

def create\_scan\_input(table\_name):

return {

"TableName": table\_name,

"ProjectionExpression": "#a97a0,#a97a1",

"ExpressionAttributeNames": {"#a97a0":"uniqueId", "#a97a1":"outputType"}

}

def parse\_dynamo\_item\_to\_dict(item):

return json.loads(json.dumps(item, indent=4, cls=DecimalEncoder))

# Helper class to convert a DynamoDB item to JSON.

class DecimalEncoder(json.JSONEncoder):

def default(self, o):

if isinstance(o, Decimal):

if o % 1 > 0:

return float(o)

else:

return int(o)

return super(DecimalEncoder, self).default(o)

import boto3

import os

import json

from decimal import Decimal

dynamodb\_resource = boto3.resource('dynamodb')

table = dynamodb\_resource.Table(os.environ["TABLE\_NAME"])

def handle(event, context):

dynamodb\_items = scan\_table()

items = [parse\_dynamo\_item\_to\_dict(item) for item in dynamodb\_items]

print(items)

trafficLocations = create\_traffic\_locations\_list(items)

response\_body = {

"trafficLocations": trafficLocations

}

response = {

"statusCode": 200,

"body": json.dumps(response\_body)

}

return response

def create\_traffic\_locations\_list(items):

print(items)

unique\_ids = list(set([d["uniqueId"] for d in items]))

trafficLocations = []

for id in unique\_ids:

data = {

id: [i for i in items if i["uniqueId"] == id]

}

trafficLocations.append(data)

return trafficLocations

def scan\_table(input={}):

response = table.scan(\*\*input)

data = response['Items']

while 'LastEvaluatedKey' in response:

response = table.scan(ExclusiveStartKey=response['LastEvaluatedKey'], \*\*input)

data.extend(response['Items'])

return data

def create\_scan\_input(table\_name):

return {

"TableName": table\_name,

"ProjectionExpression": "#a97a0,#a97a1",

"ExpressionAttributeNames": {"#a97a0": "uniqueId", "#a97a1": "outputType"}

}

def parse\_dynamo\_item\_to\_dict(item):

return json.loads(json.dumps(item, indent=4, cls=DecimalEncoder))

# Helper class to convert a DynamoDB item to JSON.

class DecimalEncoder(json.JSONEncoder):

def default(self, o):

if isinstance(o, Decimal):

if o % 1 > 0:

return float(o)

else:

return int(o)

return super(DecimalEncoder, self).default(o)

import boto3

import requests

import base64

import os

import json

STEAMING\_DATASET\_URL = os.environ.get('STEAMING\_DATASET\_URL')

def handle(event, context):

for record in event['Records']:

payload = base64.b64decode(record['kinesis']['data'])

data = json.loads(payload)

composed\_key = create\_outputType\_recordTimestamp\_composed\_key(data)

data["outputType\_recordTimestamp"] = composed\_key

print(f"Data that will be forwarded {json.dumps(data)}")

forward\_to\_pbi(data)

def create\_outputType\_recordTimestamp\_composed\_key(data\_item):

output\_type = data\_item.get('outputType')

record\_timestamp = data\_item.get('recordTimestamp')

combo\_key = output\_type + '\_' + str(record\_timestamp)

return combo\_key

def forward\_to\_pbi(data):

# data dict must be contained in a list

payload = [data]

# post/push data to the streaming API

headers = {

"Content-Type": "application/json"

}

response = requests.request(

method="POST",

url=STEAMING\_DATASET\_URL,

headers=headers,

data=json.dumps(payload)

)

print(response)

print(response.content)

import json

import boto3

import os

import dateutil.parser as dp

import logging

from collections.abc import MutableMapping

s3\_resource = boto3.resource('s3')

s3\_client = boto3.client('s3')

firehose\_client = boto3.client('firehose')

BUCKET = os.environ["BUCKET\_NAME"]

S3\_JSON\_PREFIX = os.environ["S3\_JSON\_PREFIX"]

DELIVERY\_STREAM\_NAME = os.environ["DELIVERY\_STREAM\_NAME"]

def handle(event, context):

s3\_event = event["Records"][0]

object\_key = s3\_event["s3"]["object"]["key"]

json\_data\_blob = get\_from\_s3(object\_key)

json\_data = json.loads(json\_data\_blob)

meetpunten = json\_data["miv"]["meetpunt"]

events = transform\_meetpunten\_to\_events(meetpunten)

print('Sending {} events to firehose'.format(len(events)))

push\_data\_to\_firehose(events)

response = {

"statusCode": 200,

"body": json.dumps('success!')

}

return response

def get\_from\_s3(key):

obj = s3\_resource.Object(BUCKET, key)

return obj.get()['Body'].read().decode('utf-8')

def transform\_meetpunten\_to\_events(meetpunten):

events = []

for mp in meetpunten:

event = transform\_meetpunt\_to\_event(mp)

events.append(event)

return events

def update\_event\_time(event):

current = str(event["tijd\_waarneming"])

new\_event = event.copy()

parsed\_t = dp.parse(current)

t\_in\_seconds = parsed\_t.strftime('%s')

new\_event["tijd\_waarneming"] = str(t\_in\_seconds)

return new\_event

def transform\_meetpunt\_to\_event(meetpunt):

event = meetpunt

event = clean\_keys(event)

meetdata = event["meetdata"]

meetdata\_dict = generate\_meetdata\_dict(meetdata)

event.pop("meetdata")

event.update(meetdata\_dict)

rekendata\_flattened = flatten({"rekendata": event["rekendata"]})

event.pop("rekendata")

event.update(rekendata\_flattened)

updated\_event = update\_event\_time(event)

return updated\_event

def generate\_meetdata\_dict(meetdata):

meetdata\_dict = {}

for md in meetdata:

klasse\_id = md.get("klasse\_id")

verkeersintensiteit\_key, voertuigsnelheid\_rekenkundig\_key, voertuigsnelheid\_harmonisch\_key = create\_unique\_klasse\_keys(klasse\_id)

meetdata\_dict[verkeersintensiteit\_key] = md.get('verkeersintensiteit')

meetdata\_dict[voertuigsnelheid\_rekenkundig\_key] = md.get('voertuigsnelheid\_rekenkundig')

meetdata\_dict[voertuigsnelheid\_harmonisch\_key] = md.get('voertuigsnelheid\_harmonisch')

return meetdata\_dict

def create\_unique\_klasse\_keys(klasse\_id):

key1 = 'verkeersintensiteit' + '\_klasse' + klasse\_id

key2 = 'voertuigsnelheid\_rekenkundig' + '\_klasse' + klasse\_id

key3 = 'voertuigsnelheid\_harmonisch' + '\_klasse' + klasse\_id

return key1, key2, key3

def clean\_keys(md):

md\_as\_string = json.dumps(md)

md\_as\_string\_updated = md\_as\_string.replace('@', '')

md = json.loads(md\_as\_string\_updated)

return md

def flatten(d, parent\_key='', sep='\_'):

items = []

for k, v in d.items():

new\_key = parent\_key + sep + k if parent\_key else k

if isinstance(v, MutableMapping):

items.extend(flatten(v, new\_key, sep=sep).items())

else:

items.append((new\_key, v))

return dict(items)

def push\_data\_to\_firehose(events):

number\_of\_events\_sent = 0

number\_of\_events\_to\_send = len(events)

batch = []

for i in range(number\_of\_events\_to\_send):

record = {'Data': json.dumps(events[i])}

batch.append(record)

if len(batch) == 100 or i == number\_of\_events\_to\_send - 1:

put\_batch(batch)

number\_of\_events\_sent += len(batch)

batch = []

if (i+1) % 100 == 0 or i == number\_of\_events\_to\_send - 1:

print('Number of events sent: {}'.format(number\_of\_events\_sent))

def put\_batch(batch):

response = firehose\_client.put\_record\_batch(

DeliveryStreamName=DELIVERY\_STREAM\_NAME,

Records=batch

)

import json

import boto3

import requests

import logging

import os

import time

from botocore.exceptions import ClientError

s3\_client = boto3.client('s3')

URL = os.environ["TRAFFIC\_DATA\_URL"]

BUCKET = os.environ["BUCKET\_NAME"]

S3\_XML\_PREFIX = os.environ["S3\_XML\_PREFIX"]

def retrieve\_data():

r = requests.get(URL)

return r.text

def store\_data(data, object\_key, bucket):

try:

response = s3\_client.put\_object(Body=data, Bucket=bucket, Key=object\_key, ContentType='application/xml')

except ClientError as e:

logging.error(e)

return False

return True

def handle(event, context):

xml\_data = retrieve\_data()

now = str(int(round(time.time() \* 1000)))

key = S3\_XML\_PREFIX + now + '.xml'

store\_data(xml\_data, key, BUCKET)

response = {

"statusCode": 200,

"body": 'hello'

}

return response

import boto3

import botocore

import base64

import os

import json

dynamodb\_resource = boto3.resource('dynamodb')

table = dynamodb\_resource.Table(os.environ["TABLE\_NAME"])

def handle(event, context):

print("START----------------------------------------------START")

for record in event['Records']:

payload = base64.b64decode(record['kinesis']['data'])

data = json.loads(payload)

if data.get('outputType') == 'SPEED\_DIFFERENTIAL':

save\_speed\_diff\_item(data)

elif data.get('outputType') == 'TRAFFIC\_JAM':

save\_traffic\_jam\_item(data)

elif data.get('outputType') == 'SPEED\_AVG':

item = create\_speed\_avg\_item(data)

print(f"Will save item: {item}")

table.put\_item(Item=item)

print("END----------------------------------------------END")

def save\_speed\_diff\_item(data\_item):

combo\_key = create\_outputType\_recordTimestamp\_key(data\_item)

try:

table.update\_item(

Key={

'uniqueId': str(data\_item.get('uniqueId')),

'outputType': str(data\_item.get('outputType'))

},

UpdateExpression="set recordTimestamp = :r, currentSpeed=:c, previousSpeed=:p, speedDiffIndicator=:sd, bezettingsgraad=:bg, outputType\_recordTimestamp=:otrts, loc=:l",

ExpressionAttributeValues={

':r': str(data\_item.get('recordTimestamp')),

':c': data\_item.get('currentSpeed'),

':p': data\_item.get('previousSpeed'),

':sd': data\_item.get('speedDiffIndicator'),

':bg': data\_item.get('bezettingsgraad'),

':l': data\_item.get('location'),

':otrts': combo\_key

},

ConditionExpression='speedDiffIndicator <> :sd')

except dynamodb\_resource.meta.client.exceptions.ConditionalCheckFailedException as e:

None

def create\_speed\_avg\_item(data\_item):

combo\_key = create\_outputType\_recordTimestamp\_key(data\_item)

speed\_avg\_item = {

'uniqueId': str(data\_item.get('uniqueId')),

'recordTimestamp': str(data\_item.get('recordTimestamp')),

'outputType': data\_item.get('outputType'),

'outputType\_recordTimestamp': combo\_key,

'currentSpeed': data\_item.get('currentSpeed'),

'avgSpeed3Minutes': data\_item.get('avgSpeed3Minutes'),

'avgSpeed10Minutes': data\_item.get('avgSpeed10Minutes'),

'loc': data\_item.get('location'),

}

return speed\_avg\_item

def save\_traffic\_jam\_item(data\_item):

combo\_key = create\_outputType\_recordTimestamp\_key(data\_item)

try:

table.update\_item(

Key={

'uniqueId': str(data\_item.get('uniqueId')),

'outputType': str(data\_item.get('outputType'))

},

UpdateExpression="set recordTimestamp = :r, currentSpeed=:c, trafficJamIndicator=:tji, outputType\_recordTimestamp=:otrts, loc=:l",

ExpressionAttributeValues={

':r': str(data\_item.get('recordTimestamp')),

':c': data\_item.get('currentSpeed'),

':tji': data\_item.get('trafficJamIndicator'),

':l': data\_item.get('location'),

':otrts': combo\_key

},

ConditionExpression='trafficJamIndicator <> :tji')

except dynamodb\_resource.meta.client.exceptions.ConditionalCheckFailedException as e:

None

def create\_outputType\_recordTimestamp\_key(data\_item):

output\_type = data\_item.get('outputType')

record\_timestamp = data\_item.get('recordTimestamp')

combo\_key = output\_type + '\_' + str(record\_timestamp)

return combo\_key

import json

import boto3

import os

import base64

klasses = [

"klasse1",

"klasse2",

"klasse3",

"klasse4",

"klasse5",

]

traffic\_prefixes = {

"traffic\_intensity\_prefix": "verkeersintensiteit",

"speed\_math\_prefix": "voertuigsnelheid\_rekenkundig",

"speed\_harmonic\_prefix": "voertuigsnelheid\_harmonisch"

}

def handle(event, context):

output = []

for record in event['records']:

payload = base64.b64decode(record['data'])

# print(payload.decode("utf-8"))

result = 'Dropped'

# Do custom processing on the payload here

result, updated\_payload = check\_and\_update\_payload(json.loads(payload.decode("utf-8")), result)

output\_record = {

'recordId': record['recordId'],

'result': result,

'data': base64.b64encode(json.dumps(updated\_payload).encode("utf-8")).decode("utf-8")

}

output.append(output\_record)

print('Successfully processed {} records.'.format(len(event['records'])))

return {'records': output}

def check\_and\_update\_payload(payload, result):

should\_forward = matches\_filter\_criteria(payload)

updated\_payload = None

if should\_forward:

result = 'Ok'

updated\_payload = remove\_non\_measurements(payload)

print(f"Updated payload is {updated\_payload}")

return result, updated\_payload

def matches\_filter\_criteria(payload=None):

if payload.get(traffic\_prefixes.get("speed\_harmonic\_prefix") + "\_" + klasses[1]) in ['252', '254']:

return False

filter\_ids = os.environ["FILTER\_IDS"]

if filter\_ids == '\*':

return True

else:

filter\_ids = filter\_ids.replace(" ", "")

permitted\_ids = filter\_ids.split(",")

if payload.get("unieke\_id") in permitted\_ids:

payload\_id = payload.get("unieke\_id")

print(f"Payload matched cause it has id: {payload\_id}")

print("Permitted ids are {}".format(permitted\_ids))

print("Forwarding payload")

print(json.dumps(payload))

return True

return False

def remove\_non\_measurements(payload):

for k in klasses:

harmonic\_key = traffic\_prefixes.get("speed\_harmonic\_prefix") + "\_" + k

if payload.get(harmonic\_key) == '252':

remove\_elements\_for\_klasse(payload, k)

return payload

def remove\_elements\_for\_klasse(payload, k):

for v in traffic\_prefixes.values():

payload.pop(v + '\_' + k)

import json

import boto3

import os

import xmltodict

import logging

from botocore.exceptions import ClientError

s3\_resource = boto3.resource('s3')

s3\_client = boto3.client('s3')

BUCKET = os.environ["BUCKET\_NAME"]

S3\_XML\_PREFIX = os.environ["S3\_XML\_PREFIX"]

S3\_JSON\_PREFIX = os.environ["S3\_JSON\_PREFIX"]

def handle(event, context):

s3\_event = event["Records"][0]

object\_key = s3\_event["s3"]["object"]["key"]

xml\_data = get\_from\_s3(object\_key)

json\_data = convert\_xml\_to\_dict(xml\_data)

json\_object\_key = create\_new\_key(object\_key)

store\_data(json\_data, json\_object\_key, BUCKET)

response = {

"statusCode": 200,

"body": json.dumps('success!')

}

return response

def get\_from\_s3(key):

obj = s3\_resource.Object(BUCKET, key)

return obj.get()['Body'].read().decode('utf-8')

def convert\_xml\_to\_dict(xml\_data):

my\_dict = xmltodict.parse(xml\_data)

json\_data = json.dumps(my\_dict)

return json\_data

def create\_new\_key(previous\_key):

file\_name = previous\_key.replace(S3\_XML\_PREFIX, '')

file\_name\_without\_suffix = os.path.splitext(file\_name)[0]

new\_json\_object\_key = S3\_JSON\_PREFIX + file\_name\_without\_suffix + '.json'

return new\_json\_object\_key

def store\_data(data, object\_key, bucket):

try:

response = s3\_client.put\_object(Body=data, Bucket=bucket, Key=object\_key, ContentType='application/json')

except ClientError as e:

logging.error(e)

return False

return True