**25 - R710 Proxmox Add Nomad job - localstack SQS**

These notes cover adding nomad job for:

**localstack** to provide **SQS** services,

This document builds upon the previous documents – specifically:

24 - R710 Proxmox Add 2nd disk to run host and Docker, Nomad for minio

# “localstack” install as a Nomad job to provide SQS:

1. On run3 host host, install **aws** command by doing:  
   **sudo apt install awscli**  
     
   This is needed to use / test localstack.
2. Run:  
   **aws configure**  
     
   and enter values:  
     
   AWS Access Key ID [None]: **ignore**

AWS Secret Access Key [None]: **ignore**

Default region name [None]: **us-east-1**

Default output format [None]: **json**

1. Then make sure the file: **~/.aws/config** has contents:  
   **[default]**

**region = us-east-1**

**output = json**

**[profile localstack]**

**region = us-east-1**

1. And also the file: **~/.aws/credentials** has contents:  
   **[default]**

**aws\_access\_key\_id = test**

**aws\_secret\_access\_key = test**

**[localstack]**

**aws\_access\_key\_id = test**

**aws\_secret\_access\_key = test**

If you had to change any of the above, close the terminal you made the changes via and open a new one to pick up the changes.

1. In the directory: **/mnt/S3andSQS/tmp** do:  
   **mkdir localstack**  
   **cd localstack**  
   **mkdir data**
2. In **nomad-jobs** directory create file named “**localstack-sqs.nomad**”, and put the following in it (as of 18.4.2022 this example has been made more advanced with task "create-sqs-queues" that set up SQS queues after the localstack SQS is running for use by deployer application – this has been done after stopping localstack SQS retaining created queues after reboot [by commenting out line with ‘DATA\_DIR’ in it] as localstack was logging too much when queues being used):  
   job "localstack-sqs" {

# 'datacenters' is for human convenience and has no binding significance

datacenters = ["run3"]

type = "service"

# All tasks in this job must run on client agent for host 'run3'

constraint {

attribute = "${node.class}"

value = "run3"

}

# The 'Task Group' name has the name of host 'run3' to indicate the host

# (or the type of host) this job is on:

group "run3" {

count = 1

network {

port "sqs" {

to = 4566

static = 4566

}

}

# increase default disk from 300MB to allow for log growth (for if this is left running)

ephemeral\_disk {

size = 1000

sticky = true

migrate = false

}

task "localstack-sqs" {

driver = "docker"

env {

AWS\_DEFAULT\_REGION = "us-east-1"

DEFAULT\_REGION = "us-east-1"

EDGE\_PORT = 4566

SERVICES = "sqs"

# !!! the following when used creates a huge amount of logs when the deployer is running

# showing a STUPID amount of polling or some such.

# not using it stops events being appended to 'recorded\_api\_calls.json' but also means

# queues are not retained over reboot ... !!! need to sus out whats going on in the libs

# that deployer is using.

# DATA\_DIR = "/tmp/localstack/data"

DOCKER\_HOST = "unix:///var/run/docker.sock"

}

# user = "root"

config {

#image = "localstack/localstack"

# pull specific version to stop the latest 1.5GB image being pulled every

# time i power up this environment, as force\_pull does not seem to with

# the preceding "image= ..." line.

image = "localstack/localstack:0.14.2"

force\_pull = false

ulimit {

# ensure all memory can be locked, typically of use with java apps like elastic search

memlock = "-1"

# ensure enough open file handles can be created

nofile = "65536"

# ensure enough threads can be created

#nproc = "65536"

nproc = "8192"

}

ports = ["sqs"]

logging {

# the default logger is 'json-file', which has no log-rotation,

# so change to a logging driver that does:

type = "syslog"

}

volumes = [

"/var/run/docker.sock:/var/run/docker.sock",

"/mnt/S3andSQS/tmp/localstack/data:/tmp/localstack/data"

]

}

resources {

cpu = 1000

memory = 500

}

}

# setup access to directory on host that has a 'host\_volume' section

# in '/etc/nomad.d/nomad.hcl' of:

# path = "/mnt/S3andSQS/tmp/nomad/localSQS"

volume "localSQS" {

type = "host"

read\_only = false

source = "localSQS"

}

task "create-sqs-queues" {

lifecycle {

hook = "poststart"

# This task does not seem to run properly at power up unless the task is

# configured to stick around by setting 'sidecar = true' (24.Apr.2022)

sidecar = true

}

driver = "docker"

volume\_mount {

volume = "localSQS"

# connect host directory with in container directory '/l2'

destination = "/l2"

read\_only = false

}

config {

image = "alpine:3.15.4"

force\_pull = false

ulimit {

# ensure all memory can be locked, typically of use with java apps like elastic search

memlock = "-1"

# ensure enough open file handles can be created

nofile = "65536"

# ensure enough threads can be created

#nproc = "65536"

nproc = "8192"

}

command = "sh"

# The following if for home lab testing and NOT for production as it creates keys and secrets.

# The home lab servers are not left running all the time and

# SQS queue's are not retained over reboot because of odd behavior with logging,

# so they are created after startup of the localstack SQS:

# NOTE: the following starts with a 12 second delay to ensure nomad sees this task

# as alive for >=min\_healthy\_time

# otherwise it might think the task is dead and not run it as desired.

args = ["-c", "sleep 12;apk --update --no-cache --quiet add bash jq aws-cli;/l2/connect.sh;"]

network\_mode = "host"

}

resources {

cpu = 200

memory = 256

}

}

}

}

1. In **nomad-jobs** directory, create directory: **localSQS**  
   and in directory **localSQS**, create file: **connect.sh** with the following in it (this file is used by the create-sqs-queues task):  
   #!/bin/bash

set -e # 'e' to stop on error (non zero return from script)

# NOTE: if you change this file, nomad will not automatically pick it up ...

# you have to stop and re-start the localstack-sqs Job in nomad GUI.

export AWS\_DEFAULT\_REGION=us-east-1

aws configure set aws\_access\_key\_id test --profile=default

aws configure set aws\_secret\_access\_key test --profile=default

echo -n "Waiting for 15 seconds"

# give localstack SQS a chance to get past initial startup before requesting it to make new queues

sleep 15

echo ""

echo -n "Creating queues:"

echo ""

# If this script is run from command line (outside of Nomad) then don't over-write existing queue ...

#

# NOTE:

# In Nomad for each queue that does not exist, the stderr has the following written to it:

# "An error occurred (AWS.SimpleQueueService.NonExistentQueue) when calling the GetQueueAttributes operation:

# The specified queue does not exist for this wsdl version."

result=0

aws --endpoint-url=http://192.168.124.162:4566 sqs get-queue-attributes --queue-url http://192.168.124.162:4566/000000000000/develop-deadletter-service-events --attribute-names CreatedTimestamp --region us-east-1 || result=$?

if [[ $result -ne 0 ]]; then

# create queue that does not exist

result=0

aws --endpoint-url=http://192.168.124.162:4566 sqs create-queue --queue-name develop-deadletter-service-events || result=$?

if [[ $result -ne 0 ]]; then

exit 10

fi

fi

result=0

aws --endpoint-url=http://192.168.124.162:4566 sqs get-queue-attributes --queue-url http://192.168.124.162:4566/000000000000/develop-inbound-service-events --attribute-names CreatedTimestamp --region us-east-1 || result=$?

if [[ $result -ne 0 ]]; then

# create queue that does not exist

aws --endpoint-url=http://192.168.124.162:4566 sqs create-queue --queue-name develop-inbound-service-events --attributes VisibilityTimeout=600,MessageRetentionPeriod=1209600

fi

result=0

aws --endpoint-url=http://192.168.124.162:4566 sqs get-queue-attributes --queue-url http://192.168.124.162:4566/000000000000/develop-deadletter-deployer-messages --attribute-names CreatedTimestamp --region us-east-1 || result=$?

if [[ $result -ne 0 ]]; then

# create queue that does not exist

aws --endpoint-url=http://192.168.124.162:4566 sqs create-queue --queue-name develop-deadletter-deployer-messages

fi

result=0

aws --endpoint-url=http://192.168.124.162:4566 sqs get-queue-attributes --queue-url http://192.168.124.162:4566/000000000000/develop-inbound-deployer-messages --attribute-names CreatedTimestamp --region us-east-1 || result=$?

if [[ $result -ne 0 ]]; then

# create queue that does not exist

aws --endpoint-url=http://192.168.124.162:4566 sqs create-queue --queue-name develop-inbound-deployer-messages --attributes VisibilityTimeout=600,MessageRetentionPeriod=1209600

fi

result=0

aws --endpoint-url=http://192.168.124.162:4566 sqs get-queue-attributes --queue-url http://192.168.124.162:4566/000000000000/develop-service-event-results --attribute-names CreatedTimestamp --region us-east-1 || result=$?

if [[ $result -ne 0 ]]; then

# create queue that does not exist - VisibilityTimeout=1 means release the message after one second if its not for the reader that read it.

aws --endpoint-url=http://192.168.124.162:4566 sqs create-queue --queue-name develop-service-event-results --attributes VisibilityTimeout=1,MessageRetentionPeriod=900

fi

# connect: develop-deadletter-deployer-messages

response=$(aws --endpoint-url=http://192.168.124.162:4566 sqs get-queue-attributes --queue-url http://192.168.124.162:4566/000000000000/develop-deadletter-deployer-messages --attribute-names QueueArn --region us-east-1)

if [[ -n $response ]]; then

dlq\_arn=$(echo "$response" | jq -r .Attributes.QueueArn)

echo ""

echo "The dead letter arn: $dlq\_arn"

echo ""

echo "{\"RedrivePolicy\": \"{\\\"deadLetterTargetArn\\\":\\\"$dlq\_arn\\\",\\\"maxReceiveCount\\\":\\\"1\\\"}\"}" >att.json

# Configure dead letter queue, connect it to: develop-inbound-deployer-messages

aws --endpoint-url=http://192.168.124.162:4566 sqs set-queue-attributes \

--queue-url http://192.168.124.162:4566/000000000000/develop-inbound-deployer-messages \

--region us-east-1 \

--attributes file://att.json

# Get current queue attributes for queue

aws --endpoint-url=http://192.168.124.162:4566 sqs get-queue-attributes \

--queue-url http://192.168.124.162:4566/000000000000/develop-inbound-deployer-messages \

--region us-east-1 \

--attribute-names All

fi

# connect: develop-deadletter-service-events

response=$(aws --endpoint-url=http://192.168.124.162:4566 sqs get-queue-attributes --queue-url http://192.168.124.162:4566/000000000000/develop-deadletter-service-events --attribute-names QueueArn --region us-east-1)

if [[ -n $response ]]; then

dlq\_arn=$(echo "$response" | jq -r .Attributes.QueueArn)

echo ""

echo "The dead letter arn: $dlq\_arn"

echo ""

echo "{\"RedrivePolicy\": \"{\\\"deadLetterTargetArn\\\":\\\"$dlq\_arn\\\",\\\"maxReceiveCount\\\":\\\"1\\\"}\"}" >att.json

# Configure dead letter queue, connect it to: develop-inbound-service-events

aws --endpoint-url=http://192.168.124.162:4566 sqs set-queue-attributes \

--queue-url http://192.168.124.162:4566/000000000000/develop-inbound-service-events \

--region us-east-1 \

--attributes file://att.json

# Get current queue attributes for queue

aws --endpoint-url=http://192.168.124.162:4566 sqs get-queue-attributes \

--queue-url http://192.168.124.162:4566/000000000000/develop-inbound-service-events \

--region us-east-1 \

--attribute-names All

fi

echo ""

echo -n "Finished queue setup and deadletter connections"

echo ""

# 30 Apr 2022

# In nomad, clicking on 'create-sqs-queues' and then clicking on 'Exec'

# and then adjusting the shell used to be '/bin/ash', i can get a terminal

# prompt into the alpine linux used as the 'poststart-sidecar' docker container.

# Doing 'du -h', shows ~141 Mbytes of disk used.

# Runing: 'apk del aws-cli'

# and doing 'du -h' shows disk used has dropped to ~9.3 Mbytes.

echo "current disk usage:"

du -h | tail -1 | awk '{print $1}'

echo "cleaning up space used: removing 'aws-cli'"

apk del aws-cli --quiet

echo "new disk usage:"

du -h | tail -1 | awk '{print $1}'

date

echo "spinning in loop to ensure nomad does not mark as failed"

# keep the docker container running, to ensure nomad does not mark it as failed for running to quickly

while true

do

sleep 2

done

exit 0

1. Create dir in **/mnt/S3andSQS/tmp** (as user rhys) called “**nomad**”
2. Cd into that new “nomad” dir and do:  
   **cp /home/rhys/public/nomad-jobs/localstack-sqs.nomad .**

**cp -R /home/rhys/public/nomad-jobs/localSQS/ .**

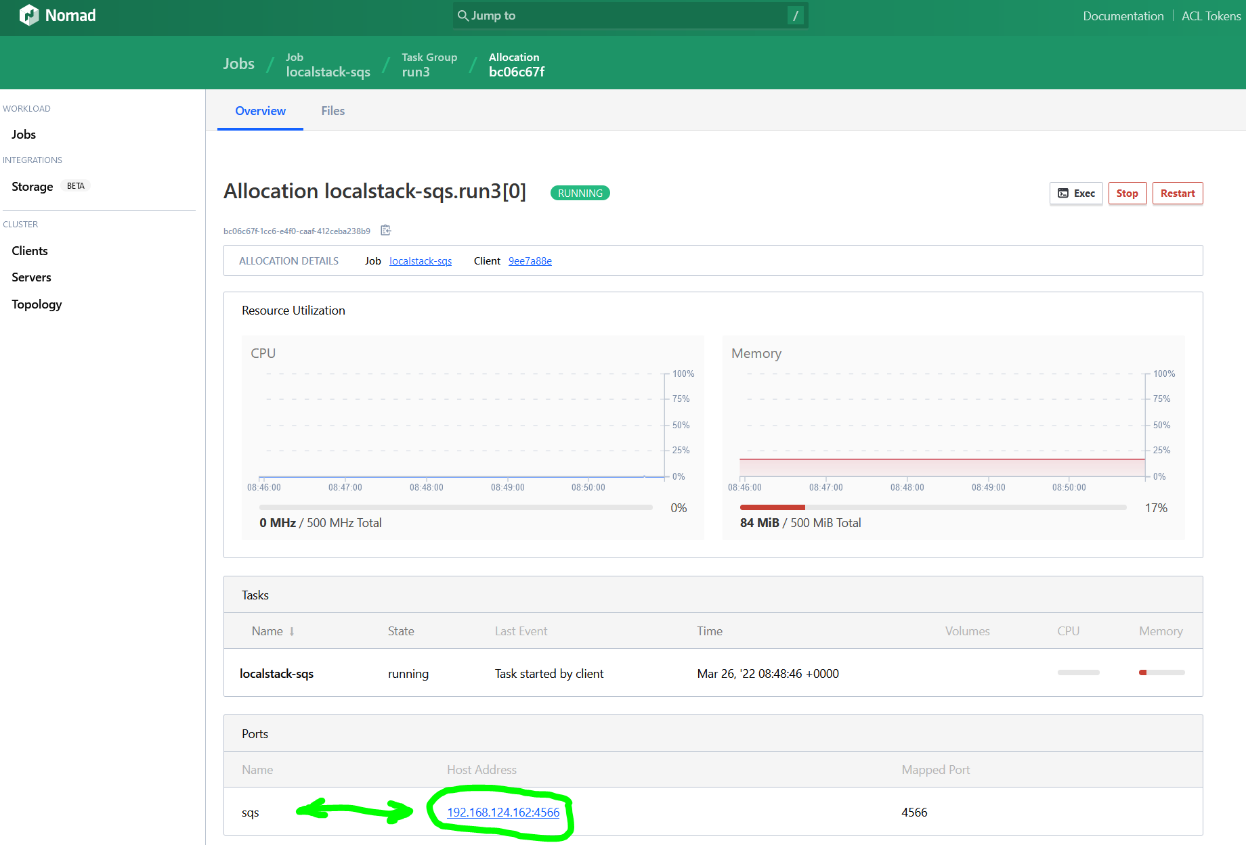
**chmod –R 700 localSQS**

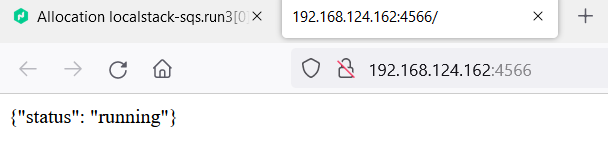
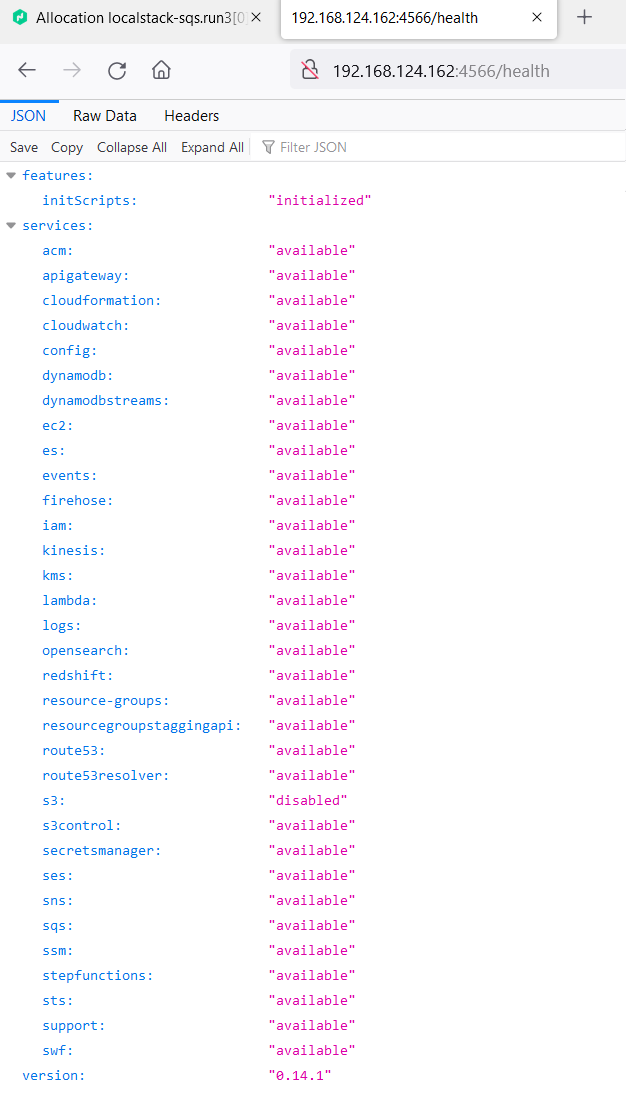
Ensure the owners and users of the nomad directory and its contents recursive is rhys:rhys

1. Check the job is ok with:  
   **nomad job plan localstack-sqs.nomad**
2. Run the job is with:  
   **nomad job run localstack-sqs.nomad**  
     
   This will take ~30 seconds to indicate OK whilst it pulls the images and runs them.
3. If you need to totally get rid of the minio job, do:

**nomad job stop -purge localstack-sqs**

1. Check in Nomad web gui that all is OK and navigate to page that looks like this and click on the **sqs** link:



1. To get:  
   
2. In the browser, add **health** on to the end of the URL and press enter to get:  
   
3. And not in the above that S3 is disabled due to minio providing S3
4. The following commands can be used to test the SQS service is operating OK:  
     
   **aws --endpoint-url=http://192.168.124.162:4566 sqs create-queue --queue-name testq**

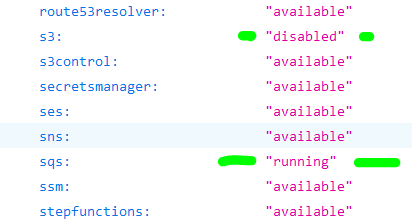
**aws --endpoint-url=http://192.168.124.162:4566 sqs list-queues**

**aws --endpoint-url=http://192.168.124.162:4566 sqs send-message --queue-url http://192.168.124.162:4566/00/testq --message-body 'wow a message'**

**aws --endpoint-url=http://192.168.124.162:4566 sqs receive-message --queue-url http://192.168.124.162:4566/00/testq**

**aws --endpoint-url=http://192.168.124.162:4566 sqs delete-queue --queue-url http://192.168.124.162:4566/00/testq**

**aws --endpoint-url=http://192.168.124.162:4566 sqs list-queues**see end of document for example of **‘get-queue-attributes’.**

1. After running these commands, going to the URL:  
   <http://192.168.124.162:4566/health>  
     
   Shows that the SQS service is now running:  
     
     
   And also in nomad the used memory has increased by ~ 16MB for the localstack-sqs job.

# ‘go’ code to test access to sqs:

On a windows system with WSL installed and goland installed in that (or another linux host with golang) …

1. On WSL system / other linux host, install **aws** command by doing:  
   **sudo apt install awscli**
2. Run:  
   **aws configure**  
     
   and enter values:  
     
   AWS Access Key ID [None]: **test**

AWS Secret Access Key [None]: **test**

Default region name [None]: **us-east-1**

Default output format [None]: **json**

1. Then make sure the file: **~/.aws/config** has contents:  
   **[default]**

**region = us-east-1**

**output = json**

**[profile localstack]**

**region = us-east-1  
output = json**

1. And also the file: **~/.aws/credentials** has contents:  
   **[localstack]**

**aws\_access\_key\_id = test**

**aws\_secret\_access\_key = test**  
  
If you had to change any of the above, close the terminal you made the changes via and open a new one to pick up the changes.

1. Create test directory called ‘**local-test**’
2. In ‘**local-test**’ create this directory tree with with the empty named files:  
     
   ├── amazon

│   ├── aws.go

│   └── sqs.go

├── cloud

│   ├── client.go

│   └── model.go

├── go.mod

├── main.go

└── message

└── message.go

1. In file ‘**main.go**’ and put the following into it:  
     
   package main

import (

"l4/amazon"

"l4/message"

"log"

"time"

)

// All credit for the localstack SQS example code goes to:

//

// http://www.inanzzz.com/index.php/post/x4cy/a-simple-aws-sqs-example-with-golang-suing-localstack

//

// from whence this code was derived.

func main() {

// Create a session instance.

ses, err := amazon.New(amazon.Config{

Address: "http://192.168.124.162:4566",

Region: "us-east-1",

Profile: "aws",

ID: "test",

Secret: "test",

})

if err != nil {

log.Fatalln(err)

}

// Test message

message.Message(amazon.NewSQS(ses, time.Second\*5))

}

1. In file ‘**message.go**’ and put the following into it:  
     
   package message

import (

"context"

"l4/cloud"

"log"

)

func Message(client cloud.MessageClient) {

ctx := context.Background()

dlxURL := createQueueDLX(ctx, client)

queURL := createQueue(ctx, client)

dlxARN := queueARN(ctx, client, dlxURL)

bindDLX(ctx, client, queURL, dlxARN)

send(ctx, client, queURL)

rcvHnd := receive(ctx, client, queURL)

deleteMessage(ctx, client, queURL, rcvHnd)

}

func createQueueDLX(ctx context.Context, client cloud.MessageClient) string {

url, err := client.CreateQueue(ctx, "welcome-email-queue.dlx", true)

if err != nil {

log.Fatalln(err)

}

log.Println("create queue:", url)

return url

}

func createQueue(ctx context.Context, client cloud.MessageClient) string {

url, err := client.CreateQueue(ctx, "welcome-email-queue", false)

if err != nil {

log.Fatalln(err)

}

log.Println("create queue:", url)

return url

}

func queueARN(ctx context.Context, client cloud.MessageClient, url string) string {

arn, err := client.QueueARN(ctx, url)

if err != nil {

log.Fatalln(err)

}

log.Println("queue ARN:", arn)

return arn

}

func bindDLX(ctx context.Context, client cloud.MessageClient, queueURL, dlxARN string) {

if err := client.BindDLX(ctx, queueURL, dlxARN); err != nil {

log.Fatalln(err)

}

log.Println("bind DLX: ok")

}

func send(ctx context.Context, client cloud.MessageClient, queueURL string) {

id, err := client.Send(ctx, &cloud.SendRequest{

QueueURL: queueURL,

Body: "Message body!",

Attributes: []cloud.Attribute{

{

Key: "Title",

Value: "SQS send message",

Type: "String",

},

{

Key: "Year",

Value: "2021",

Type: "Number",

},

},

})

if err != nil {

log.Fatalln(err)

}

log.Println("send: message ID:", id)

}

func receive(ctx context.Context, client cloud.MessageClient, queueURL string) string {

res, err := client.Receive(ctx, queueURL)

if err != nil {

log.Fatalln(err)

}

log.Println("receive:", res)

return res.ReceiptHandle

}

func deleteMessage(ctx context.Context, client cloud.MessageClient, queueURL, rcvHnd string) {

if err := client.Delete(ctx, queueURL, rcvHnd); err != nil {

log.Fatalln(err)

}

log.Println("delete message: ok")

}

1. In file ‘**go.mod**’ and put the following into it:  
     
   module l4

go 1.18

require github.com/aws/aws-sdk-go v1.43.26

require github.com/jmespath/go-jmespath v0.4.0 // indirect

1. In file ‘**model.go**’ and put the following into it:  
     
   package cloud

type SendRequest struct {

QueueURL string

Body string

Attributes []Attribute

}

type Attribute struct {

Key string

Value string

Type string

}

type Message struct {

ID string

ReceiptHandle string

Body string

Attributes map[string]string

}

1. In file ‘**client.go**’ and put the following into it:  
     
   package cloud

import (

"context"

)

type MessageClient interface {

// Creates a new long polling queue and returns its URL.

CreateQueue(ctx context.Context, queueName string, isDLX bool) (string, error)

// Get a queue ARN.

QueueARN(ctx context.Context, queueURL string) (string, error)

// Binds a DLX queue to a normal queue.

BindDLX(ctx context.Context, queueURL, dlxARN string) error

// Send a message to queue and returns its message ID.

Send(ctx context.Context, req \*SendRequest) (string, error)

// Long polls given amount of messages from a queue.

Receive(ctx context.Context, queueURL string) (\*Message, error)

// Deletes a message from a queue.

Delete(ctx context.Context, queueURL, rcvHandle string) error

}

1. In file ‘**sqs.go**’ and put the following into it:  
     
   package amazon

import (

"context"

"encoding/json"

"fmt"

"l4/cloud"

"time"

"github.com/aws/aws-sdk-go/aws"

"github.com/aws/aws-sdk-go/aws/session"

"github.com/aws/aws-sdk-go/service/sqs"

)

var \_ cloud.MessageClient = SQS{}

type SQS struct {

timeout time.Duration

client \*sqs.SQS

}

func NewSQS(session \*session.Session, timeout time.Duration) SQS {

return SQS{

timeout: timeout,

client: sqs.New(session),

}

}

func (s SQS) CreateQueue(ctx context.Context, queueName string, isDLX bool) (string, error) {

ctx, cancel := context.WithTimeout(ctx, s.timeout)

defer cancel()

ret := "345600" // 4 days

if isDLX {

ret = "1209600" // 14 days

}

res, err := s.client.CreateQueueWithContext(ctx, &sqs.CreateQueueInput{

QueueName: aws.String(queueName),

Attributes: map[string]\*string{

"MessageRetentionPeriod": aws.String(ret),

"VisibilityTimeout": aws.String("5"),

"ReceiveMessageWaitTimeSeconds": aws.String("20"), // Enable long polling

},

})

if err != nil {

return "", fmt.Errorf("create: %w", err)

}

return \*res.QueueUrl, nil

}

func (s SQS) QueueARN(ctx context.Context, queueURL string) (string, error) {

ctx, cancel := context.WithTimeout(ctx, s.timeout)

defer cancel()

res, err := s.client.GetQueueAttributesWithContext(ctx, &sqs.GetQueueAttributesInput{

AttributeNames: []\*string{aws.String("QueueArn")},

QueueUrl: aws.String(queueURL),

})

if err != nil {

return "", fmt.Errorf("get attributes: %w", err)

}

if len(res.Attributes) != 1 {

return "", fmt.Errorf("not found")

}

return \*res.Attributes["QueueArn"], nil

}

func (s SQS) BindDLX(ctx context.Context, queueURL, dlxARN string) error {

ctx, cancel := context.WithTimeout(ctx, s.timeout)

defer cancel()

policy, err := json.Marshal(map[string]string{

"deadLetterTargetArn": dlxARN,

"maxReceiveCount": "3",

})

if err != nil {

return fmt.Errorf("marshal policy")

}

if \_, err := s.client.SetQueueAttributesWithContext(ctx, &sqs.SetQueueAttributesInput{

QueueUrl: aws.String(queueURL),

Attributes: map[string]\*string{

sqs.QueueAttributeNameRedrivePolicy: aws.String(string(policy)),

},

}); err != nil {

return fmt.Errorf("set attributes: %w", err)

}

return nil

}

func (s SQS) Send(ctx context.Context, req \*cloud.SendRequest) (string, error) {

ctx, cancel := context.WithTimeout(ctx, s.timeout)

defer cancel()

attrs := make(map[string]\*sqs.MessageAttributeValue, len(req.Attributes))

for \_, attr := range req.Attributes {

attrs[attr.Key] = &sqs.MessageAttributeValue{

StringValue: aws.String(attr.Value),

DataType: aws.String(attr.Type),

}

}

res, err := s.client.SendMessageWithContext(ctx, &sqs.SendMessageInput{

MessageAttributes: attrs,

MessageBody: aws.String(req.Body),

QueueUrl: aws.String(req.QueueURL),

})

if err != nil {

return "", fmt.Errorf("send: %w", err)

}

return \*res.MessageId, nil

}

func (s SQS) Receive(ctx context.Context, queueURL string) (\*cloud.Message, error) {

// timeout = WaitTimeSeconds + 5

ctx, cancel := context.WithTimeout(ctx, time.Second\*(20+5))

defer cancel()

res, err := s.client.ReceiveMessageWithContext(ctx, &sqs.ReceiveMessageInput{

QueueUrl: aws.String(queueURL),

MaxNumberOfMessages: aws.Int64(1),

WaitTimeSeconds: aws.Int64(20),

MessageAttributeNames: aws.StringSlice([]string{"All"}),

})

if err != nil {

return nil, fmt.Errorf("receive: %w", err)

}

if len(res.Messages) == 0 {

return nil, nil

}

attrs := make(map[string]string)

for key, attr := range res.Messages[0].MessageAttributes {

attrs[key] = \*attr.StringValue

}

return &cloud.Message{

ID: \*res.Messages[0].MessageId,

ReceiptHandle: \*res.Messages[0].ReceiptHandle,

Body: \*res.Messages[0].Body,

Attributes: attrs,

}, nil

}

func (s SQS) Delete(ctx context.Context, queueURL, rcvHandle string) error {

ctx, cancel := context.WithTimeout(ctx, s.timeout)

defer cancel()

if \_, err := s.client.DeleteMessageWithContext(ctx, &sqs.DeleteMessageInput{

QueueUrl: aws.String(queueURL),

ReceiptHandle: aws.String(rcvHandle),

}); err != nil {

return fmt.Errorf("delete: %w", err)

}

return nil

}

1. In file ‘**aws.go**’ and put the following into it:  
     
   package amazon

import (

"github.com/aws/aws-sdk-go/aws"

"github.com/aws/aws-sdk-go/aws/credentials"

"github.com/aws/aws-sdk-go/aws/session"

)

type Config struct {

Address string

Region string

Profile string

ID string

Secret string

}

func New(config Config) (\*session.Session, error) {

return session.NewSessionWithOptions(

session.Options{

Config: aws.Config{

Credentials: credentials.NewStaticCredentials(config.ID, config.Secret, ""),

Region: aws.String(config.Region),

Endpoint: aws.String(config.Address),

S3ForcePathStyle: aws.Bool(true),

},

Profile: config.Profile,

},

)

}

1. Then in ‘**local-test**’ do:  
     
   **go mod tidy  
     
   go run main.go**to see result similar to:  
     
   *2022/03/26 14:56:11 create queue: http://localhost:4566/000000000000/welcome-email-queue.dlx*

*2022/03/26 14:56:11 create queue: http://localhost:4566/000000000000/welcome-email-queue*

*2022/03/26 14:56:11 queue ARN: arn:aws:sqs:us-east-1:000000000000:welcome-email-queue.dlx*

*2022/03/26 14:56:11 bind DLX: ok*

*2022/03/26 14:56:11 send: message ID: 8691e909-23a6-ef7f-26c1-16800bbdbcf6*

*2022/03/26 14:56:11 receive: &{8691e909-23a6-ef7f-26c1-16800bbdbcf6 ygshwtldsxfliarsqvplsemzaazajzlcajjnomdctqgjatlwcxfqsnvbnqfxgcjqkxfdezbhjgbeqeqlsqsmxbxhbxppersblbazkaccdlqpsevqkyjfgjrwowomusoqhdartpezvhsjriambgnekgvyyjbmeqfvotzpqiisdfnegltdkkfnpomqm Message body! map[Title:SQS send message Year:2021]}*

*2022/03/26 14:56:11 delete message: ok*

1. Then in terminal on **run3** host, do:  
     
   **aws --endpoint-url=http://192.168.124.162:4566 sqs list-queues**  
     
   to see:  
     
   *{*

*"QueueUrls": [*

*"http://localhost:4566/000000000000/welcome-email-queue.dlx",*

*"http://localhost:4566/000000000000/welcome-email-queue"*

*]*

*}*

1. Then in terminal on **run3** host, do:  
     
   **aws --endpoint-url=http://192.168.124.162:4566 sqs get-queue-attributes --queue-url http://192.168.124.162:4566/000000000000/welcome-email-queue --attribute-names All**  
     
   to see:  
     
   *{*

*"Attributes": {*

*"ApproximateNumberOfMessages": "1",*

*"ApproximateNumberOfMessagesDelayed": "0",*

*"ApproximateNumberOfMessagesNotVisible": "0",*

*"CreatedTimestamp": "1648389353.724997",*

*"DelaySeconds": "0",*

*"LastModifiedTimestamp": "1648389353.760278",*

*"MaximumMessageSize": "262144",*

*"MessageRetentionPeriod": "345600",*

*"QueueArn": "arn:aws:sqs:us-east-1:000000000000:welcome-email-queue",*

*"RedrivePolicy": "{\"deadLetterTargetArn\":\"arn:aws:sqs:us-east-1:000000000000:welcome-email-queue.dlx\",\"maxReceiveCount\":3}",*

*"ReceiveMessageWaitTimeSeconds": "20",*

*"VisibilityTimeout": "5"*

*}*

*}*

1. That completes the setup and testing **localstack SQS**