#Author : Authors of Learn & Practice AI by Projects

#Purpose : RabbitMQ worker application to process the request from the RabbitMQ.

#Implements the functionality of worker to process the prediction model requests.

#Worker will listen to the incoming request queue and process the request(s) once the request is available.

#Upon processing the request, worker will send the response to the server application.

#import libraries

import pika

import time, json

import joblib

import requests

#define keys/shared names

PREDICTION\_RESULT\_KEY = 'PREDICTION\_RESULT';

SHARED\_REQUEST\_QUEUE = 'SHARED\_REQUEST\_CLASSIFICATION\_MODEL\_QUEUE';

#Constant string for Job Id and prediction request parameters.

JOBID\_KEY = 'JOBID';

SEPAL\_LENGTH = 'SEPAL\_LENGTH';

SEPAL\_WIDTH = 'SEPAL\_WIDTH'

PETAL\_LENGTH = 'PETAL\_LENGTH';

PETAL\_WIDTH = 'PETAL\_WIDTH';

#model file name

joblib\_file = "Iris\_Classifer.jbl";

#load the model file from the disk for prediction

classification\_model = joblib.load(joblib\_file);

#server API URL to update the prediction result

SERVER\_PREDICTION\_RESULT\_UPDATE\_URL = 'http://127.0.0.1:8000/update-predicted-result';

#construct the json response using given job id and prediction result

def getPredictionJSONResponse(job\_id, prediction\_result):

data ={};

data[JOBID\_KEY] = job\_id;

data[PREDICTION\_RESULT\_KEY] = prediction\_result;

jsonData = json.dumps(data);

print(jsonData);

return jsonData;

#extract the request and return as list.

def getPredictionRequest(jsonRequest):

jsonRequestObject = json.loads(jsonRequest);

requestList = [];

requestList.append(jsonRequestObject.get(SEPAL\_LENGTH));

requestList.append(jsonRequestObject.get(SEPAL\_WIDTH));

requestList.append(jsonRequestObject.get(PETAL\_LENGTH));

requestList.append(jsonRequestObject.get(PETAL\_WIDTH));

jobId = jsonRequestObject.get(JOBID\_KEY);

return jobId, [requestList];

#connection parameters required to connect to RabbitMQ

requestConnectionParameters = pika.ConnectionParameters(host='localhost');

#BlockingConnection creates layer on top of Pika's Async core to return response.

requestConnection = pika.BlockingConnection(requestConnectionParameters);

requestChannel = requestConnection.channel();

#define the request queue

requestChannel.queue\_declare(queue=SHARED\_REQUEST\_QUEUE, durable=True);

print(' [\*\*\*\*] Waiting for messages. To exit press CTRL+C')

def requestCallback(ch, method, properties, body):

receivedRequest = body.decode();

print("<-<-<- Received %r" % receivedRequest);

# sleep 5 seconds.

time.sleep(5);

jobId, prediction\_reqeust = getPredictionRequest(receivedRequest);

print(prediction\_reqeust);

prediction\_result = classification\_model.predict(prediction\_reqeust);

print('prediction result : {}'.format(prediction\_result));

jonResponse = getPredictionJSONResponse(jobId, prediction\_result[0]);

payload = {'job\_id': jobId, 'predicted\_result': prediction\_result[0]}

response = requests.get(SERVER\_PREDICTION\_RESULT\_UPDATE\_URL, params=payload)

print('Updated result successfully. status : {}'.format(response.json()));

print(" [\*\*\*\*] Processed the request for Job Id {}. Worker is ready to pick the next task.".format(jobId));

return requestChannel.basic\_ack(delivery\_tag=method.delivery\_tag)

requestChannel.basic\_qos(prefetch\_count=1)

requestChannel.basic\_consume(queue=SHARED\_REQUEST\_QUEUE, on\_message\_callback=requestCallback)

requestChannel.start\_consuming()