rabbitmq>amqp\_lib.py, amqp\_setup.py

amqp\_lib.py:

"""

Reusable AMQP-related functions

References:

https://pika.readthedocs.io/en/stable/\_modules/pika/exceptions.html#ConnectionClosed

"""

import time

import pika

def connect(hostname, port, exchange\_name, exchange\_type, max\_retries=12, retry\_interval=5,):

retries = 0

# loop to retry connection up to 12 times

# with a retry interval of 5 seconds

while retries < max\_retries:

retries += 1

try:

print(f"Connecting to AMQP broker {hostname}:{port}...")

# connect to the broker

connection = pika.BlockingConnection(

pika.ConnectionParameters(

host=hostname,

port=port,

heartbeat=300,

blocked\_connection\_timeout=300,

)

)

print("Connected")

print("Open channel")

channel = connection.channel()

# Check whether the exchange exists

print(f"Check existence of exchange: {exchange\_name}")

channel.exchange\_declare(

exchange=exchange\_name,

exchange\_type=exchange\_type,

passive=True,

)

# passive=True: If exchange does not exist, raise an error.

print("Connected")

return connection, channel

except pika.exceptions.ChannelClosedByBroker as exception:

message = f"{exchange\_type} exchange {exchange\_name} not found."

connection.close()

raise Exception(message) from exception

except pika.exceptions.AMQPConnectionError as exception:

print(f"Failed to connect: {exception=}")

print(f"Retrying in {retry\_interval} seconds...")

time.sleep(retry\_interval)

raise Exception(f"Max {max\_retries} retries exceeded...")

def close(connection, channel):

channel.close()

connection.close()

def is\_connection\_open(connection):

try:

connection.process\_data\_events()

return True

except pika.exceptions.AMQPError as e:

print("AMQP Error:", e)

return False

def start\_consuming(

hostname, port, exchange\_name, exchange\_type, queue\_name, callback

):

while True:

try:

connection, channel = connect(

hostname=hostname,

port=port,

exchange\_name=exchange\_name,

exchange\_type=exchange\_type,

)

print(f"Consuming from queue: {queue\_name}")

channel.basic\_consume(

queue=queue\_name, on\_message\_callback=callback, auto\_ack=True

)

channel.start\_consuming()

except pika.exceptions.ChannelClosedByBroker as exception:

message = f"Queue {queue\_name} not found."

connection.close()

raise Exception(message) from exception

except pika.exceptions.ConnectionClosedByBroker:

print("Connection closed. Try to reconnect...")

continue

except KeyboardInterrupt:

close(connection, channel)

break

# Other types of exception are passed on to caller to handle.

# Most likely, system issue - RabbitMQ host overload.

amqp\_setup.py:

#!/usr/bin/env python3

"""

A standalone script to create exchanges and queues on RabbitMQ.

"""

import pika

amqp\_host = "localhost"

amqp\_port = 5672

exchange\_name = "notification\_topic"

exchange\_type = "topic"

def create\_exchange(hostname, port, exchange\_name, exchange\_type):

print(f"Connecting to AMQP broker {hostname}:{port}...")

# connect to the broker

connection = pika.BlockingConnection(

pika.ConnectionParameters(

host=hostname,

port=port,

heartbeat=300,

blocked\_connection\_timeout=300,

)

)

print("Connected")

print("Open channel")

channel = connection.channel()

# Set up the exchange if the exchange doesn't exist

print(f"Declare exchange: {exchange\_name}")

channel.exchange\_declare(

exchange=exchange\_name, exchange\_type=exchange\_type, durable=True

)

# 'durable' makes the exchange survive broker restarts

return channel

def create\_queue(channel, exchange\_name, queue\_name, routing\_key):

print(f"Bind to queue: {queue\_name}")

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

# 'durable' makes the queue survive broker restarts

# bind the queue to the exchange via the routing\_key

channel.queue\_bind(

exchange=exchange\_name, queue=queue\_name, routing\_key=routing\_key

)

channel = create\_exchange(

hostname=amqp\_host,

port=amqp\_port,

exchange\_name=exchange\_name,

exchange\_type=exchange\_type,

)

create\_queue(

channel=channel,

exchange\_name=exchange\_name,

queue\_name="Order\_Confirmation",

routing\_key="order.confirmation",

)

create\_queue(

channel=channel,

exchange\_name=exchange\_name,

queue\_name="Reservation\_Confirmation",

routing\_key="reservation.confirmation",

)

create\_queue(

channel=channel,

exchange\_name=exchange\_name,

queue\_name="Reservation\_Cancellation",

routing\_key="reservation.cancellation",

)

create\_queue(

channel=channel,

exchange\_name=exchange\_name,

queue\_name="Reallocation\_Notice",

routing\_key="reallocation.notice",

)

create\_queue(

channel=channel,

exchange\_name=exchange\_name,

queue\_name="Reallocation\_Confirmation",

routing\_key="reallocation.confirmation",

)

services>common, complex, simple

complex>cancel\_booking.py

cancel\_booking.py:

from flask import Flask, request, jsonify

from flask\_sqlalchemy import SQLAlchemy

from flask\_cors import CORS

from datetime import datetime, timezone

from simple.reservation import Reservation

from common.user import User

from common.db import db

import sys, os, json, pika, rabbitmq.amqp\_lib, time, rabbitmq.amqp\_setup

app = Flask(\_\_name\_\_)

CORS(app)

db.init\_app(app)

connection = None

channel = None

def connectAMQP():

    """ Establish connection to RabbitMQ using amqp\_lib """

    global connection, channel

    max\_retries = 5

    for attempt in range(max\_retries):

        try:

            if connection is None or not rabbitmq.amqp\_lib.is\_connection\_open(connection):

                print("  Connecting to AMQP broker...")

                connection, channel = rabbitmq.amqp\_lib.connect(

                    hostname=rabbitmq.amqp\_setup.amqp\_host,

                    port=rabbitmq.amqp\_setup.amqp\_port,

                    exchange\_name=rabbitmq.amqp\_setup.exchange\_name,

                    exchange\_type=rabbitmq.amqp\_setup.exchange\_type,

                )

            return

        except Exception as e:

            print(f"  Attempt {attempt+1}/{max\_retries}: Unable to connect to RabbitMQ: {e}")

            if attempt < max\_retries - 1:

                time.sleep(2)

            else:

                print("  Max retries reached, exiting...")

                exit(1)

def publish\_message(message):

    """ Publish message to RabbitMQ """

    if connection is None or not rabbitmq.amqp\_lib.is\_connection\_open(connection):

        connectAMQP()

    message\_json = json.dumps(message)

    rabbitmq.amqp\_setup.channel.basic\_publish(

        exchange=rabbitmq.amqp\_setup.exchange\_name,

        routing\_key="reservation.cancellation",

        body=message\_json

    )

    print(f"  Published message: {message\_json}")

@app.route('/reservation/<int:reservation\_id>', methods=['PATCH'])

def update\_reservation(reservation\_id):

    #Updates the reservation and triggers a notification message via RabbitMQ

    reservation = Reservation.query.get(reservation\_id)

    if reservation is None:

        return jsonify({"message": "Reservation not found"}), 404

    user= User.query.get(reservation.user\_id)

    if user is None:

        return jsonify({"message": "User not found"}), 404

    refund\_amount = reservation.price

    try:

        # Update reservation details

        reservation.user\_id = None

        reservation.status = 'empty'

        reservation.count = None

        reservation.price = None

        reservation.time = datetime.now(timezone.utc)

        db.session.commit()

    except Exception as e:

        print(f"Database error: {e}")

        return jsonify({"error": "Database update failed", "details": str(e)}), 500

    # Prepare notification message

    notification\_data = {

        "reservation\_id": reservation\_id,

        "user\_name": user.name,

        "user\_phone": user.phone,

        "refund\_amount": refund\_amount,

        "message\_type": "reservation.cancellation"

    }

    try:

        publish\_message(notification\_data)

        return jsonify({

            "message": "Reservation cancelled and notification service triggered.",

            "status": reservation.status,

            "user\_id": reservation.user\_id,

            "count": reservation.count,

            "price": reservation.price,

            "time": reservation.time.isoformat()

        }), 200

    except Exception as e:

        return jsonify({"error": f"Reservation updated, but error triggering notification: {str(e)}"}), 505

if \_\_name\_\_ == '\_\_main\_\_':

    print("Starting cancel\_booking service...")

    connectAMQP()

    app.run(host='0.0.0.0', port=5002, debug=True)

simple>notification.py, reservation.py

notification.py:

import os, json, threading, pika, rabbitmq.amqp\_setup, rabbitmq.amqp\_lib, time

from flask import Flask, jsonify, request

from flask\_sqlalchemy import SQLAlchemy

from flask\_cors import CORS

from dotenv import load\_dotenv

from twilio.rest import Client

from datetime import datetime

# Load environment variables

load\_dotenv()

app = Flask(\_\_name\_\_)

CORS(app)

# Database configuration

app.config['SQLALCHEMY\_DATABASE\_URI'] = os.getenv('DATABASE\_URI2')

app.config['SQLALCHEMY\_TRACK\_MODIFICATIONS'] = False

db = SQLAlchemy(app)

# Twilio & RabbitMQ configuration

TWILIO\_ACCOUNT\_SID = os.getenv('TWILIO\_ACCOUNT\_SID')

TWILIO\_AUTH\_TOKEN = os.getenv('TWILIO\_AUTH\_TOKEN')

TWILIO\_PHONE\_NUMBER = os.getenv('TWILIO\_PHONE\_NUMBER')

# Twilio client

client = Client(TWILIO\_ACCOUNT\_SID, TWILIO\_AUTH\_TOKEN)

# Notification model

class Notification(db.Model):

    \_\_tablename\_\_ = 'notification'

    notification\_id = db.Column(db.Integer, primary\_key=True)

    message = db.Column(db.String(999), nullable=False)

    status = db.Column(db.Boolean, default=False)  # True if sent, False if failed

    type = db.Column(db.String(999), nullable=False)

    time = db.Column(db.DateTime, default=datetime.utcnow)

# Message templates for different event types

MESSAGE\_TEMPLATES = {

    "reservation.cancellation": "Hi there {username}! Your reservation for {reservation\_id} has been canceled and a refund of {refund\_amount} has been processed. We look forward to seeing you again! Thank you!",

    "order.confirmation": "Your order has been confirmed. Thank you for dining with us!",

    "reservation.confirmation": "Your reservation for {reservation\_id} has been confirmed. See you soon!",

    "reallocation.notice": "Hi there {username}! Table {table\_no} is currently open, would you like to book it? If so, please tap on the SMS to start the booking process...",

    "reallocation.confirmation": "Table {table\_no} booking has been confirmed. Thank you!"

}

#Sends an SMS via Twilio

def send\_sms(phone, message):

    try:

        sms = client.messages.create(

            body=message,

            from\_=TWILIO\_PHONE\_NUMBER,

            to=phone

        )

        return {"status": "success", "twilio\_sid": sms.sid}

    except Exception as e:

        return {"status": "failed", "error": str(e)}

#Saves the notification to the database

def save\_notification\_to\_db(message, msg\_type, status):

    new\_notification = Notification(

        message=message,

        status=status,

        type=msg\_type

    )

    db.session.add(new\_notification)

    db.session.commit()

def rabbitmq\_callback(ch, method, properties, body):

    try:

        data = json.loads(body)

        msg\_type = data.get("message\_type")

        user\_phone = data.get("user\_phone")

        username = data.get("user\_name", "there")  #return there as the name if name is empty

        reservation\_id = data.get("reservation\_id", "N/A") #return NA if id is empty

        refund\_amount = data.get("refund\_amount", "N/A") #return NA if amount is empty

        if not user\_phone or not msg\_type:

            print("Missing required fields in RabbitMQ message")

            return

        # Format the message based on the event type

        message\_template = MESSAGE\_TEMPLATES.get(msg\_type, "Notification: {msg\_type}")

        formatted\_message = message\_template.format(

            username=username,

            reservation\_id=reservation\_id,

            refund\_amount=refund\_amount,

            new\_table\_no=data.get("new\_table\_no", "N/A")

        )

        print(f"Processing {msg\_type} event...")

        sms\_result = send\_sms(user\_phone, formatted\_message)

        # Save the notification to the database

        save\_notification\_to\_db(formatted\_message, msg\_type, status=(sms\_result["status"] == "success"))

        if sms\_result["status"] == "success":

            print(f"Notification sent successfully for {msg\_type}")

        else:

            print(f"Failed to send notification for {msg\_type}: {sms\_result['error']}")

    except Exception as e:

        print(f"Error processing RabbitMQ message: {e}")

def start\_rabbitmq\_consumer():

    while True:

        try:

            print("Connecting to RabbitMQ...")

            connection, channel = rabbitmq.amqp\_lib.connect(

                hostname=rabbitmq.amqp\_setup.amqp\_host,

                port=rabbitmq.amqp\_setup.amqp\_port,

                exchange\_name=rabbitmq.amqp\_setup.exchange\_name,

                exchange\_type=rabbitmq.amqp\_setup.exchange\_type,

            )

            queues = {

                "Order\_Confirmation": "order.confirmation",

                "Reservation\_Confirmation": "reservation.confirmation",

                "Reservation\_Cancellation": "reservation.cancellation",

                "Reallocation\_Notice": "reallocation.notice",

                "Reallocation\_Confirmation": "reallocation.confirmation",

            }

            for queue\_name, routing\_key in queues.items():

                print(f"Consuming from queue: {queue\_name}")

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

                channel.queue\_bind(

                    exchange=rabbitmq.amqp\_setup.exchange\_name,

                    queue=queue\_name,

                    routing\_key=routing\_key,

                )

                channel.basic\_consume(

                    queue=queue\_name,

                    on\_message\_callback=rabbitmq\_callback,

                    auto\_ack=True,

                )

            print("Waiting for messages...")

            channel.start\_consuming()

        except pika.exceptions.ConnectionClosedByBroker:

            print("Connection closed by broker. Reconnecting...")

            continue

        except KeyboardInterrupt:

            print("Stopping RabbitMQ consumer...")

            break

        except Exception as e:

            print(f"Unexpected error in RabbitMQ consumer: {e}")

            time.sleep(5)

# Health check route

@app.route('/health', methods=['GET'])

def health\_check():

    return jsonify({"status": "Notification service running"}), 200

if \_\_name\_\_ == '\_\_main\_\_':

    # Start RabbitMQ consumer in a separate thread

    threading.Thread(target=start\_rabbitmq\_consumer, daemon=True).start()

    app.run(host='0.0.0.0', port=5003, debug=True)

reservation.py:

import os

from flask import Flask, request, jsonify

from flask\_sqlalchemy import SQLAlchemy

from flask\_cors import CORS

from dotenv import load\_dotenv

from common.db import db

from common.user import User

from restaurant import Restaurant

load\_dotenv()

app = Flask(\_\_name\_\_)

CORS(app)

#Database configuration

app.config['SQLALCHEMY\_DATABASE\_URI'] = os.getenv('DATABASE\_URI')

app.config['SQLALCHEMY\_TRACK\_MODIFICATIONS'] = False

db.init\_app(app)

#Reservation Model

class Reservation(db.Model):

    \_\_tablename\_\_ = 'reservation'

    reservation\_id = db.Column(db.Integer, primary\_key=True)

    restaurant\_id = db.Column(db.Integer, nullable=False)

    user\_id = db.Column(db.Integer, nullable=True)

    table\_no = db.Column(db.Integer, nullable=True)

    status = db.Column(db.String(999), nullable=False)

    count = db.Column(db.Integer, nullable=True)

    price = db.Column(db.Float(precision=2), nullable=True)

    time = db.Column(db.DateTime, nullable=True)

    user = db.relationship('User', back\_populates='reservations', foreign\_keys=[user\_id])

    restaurant = db.relationship('Restaurant', back\_populates='reservations', foreign\_keys=[restaurant\_id])

#Retrieving of specific reservation via id

@app.route('/reservation/<int:reservation\_id>', methods=['GET'])

def get\_reservation(reservation\_id):

    reservation = Reservation.query.get(reservation\_id)

    if reservation is None:

        return jsonify({"message": "Reservation not found"}), 404

    return jsonify({

        "reservation\_id": reservation.reservation\_id,

        "restaurant\_id": reservation.restaurant\_id,

        "user\_id": reservation.user\_id,

        "table\_no": reservation.table\_no,

        "status": reservation.status,

        "count": reservation.count,

        "price": reservation.price,

        "time": reservation.time.isoformat() if reservation.time else None

    })

if \_\_name\_\_ == '\_\_main\_\_':

    print("Starting reservation service...")

    app.run(host='0.0.0.0', port=5001, debug=True)

scenario:

provide me with the reallocation\_reservation.py code along with modifying the cancel\_booking.py code (if necessary)

this code should initiate the moment the notification is being sent to confirm the reservation.cancellation. the cancel\_booking.py will pass the reservation\_id of the cancelled table to the reallocation\_reservation. The reallocation\_reservation is a composite microservice that needs to interact with the waitlist microservice (hosted in outsystem). Based on the reservation\_id given. the composite ms should assign a user that is on the priority queue/waitlist (stored in the waitlist database) to the empty/cancelled reservation table. Upon assigning, the status within the reservation data table should be updated to “pending” and the service should queue a message to rabbit mq following the format of:  
 "reservation\_id": reservation\_id,

        "user\_name": user.name,

        "user\_phone": user.phone

“table\_no”: table\_no

        "message\_type": "reallocation.notice"