Mq Push

public Mono<String> sendMessageToMq(MqConfiguration mqConfiguration, String messageToSend, String queueName, MQQueueManager qMgr,String uniqueKey,OutgoingMqMessage outgoingMqMessage)  
{  
 *log*.info("Send Message To Queue {} Method Started with Retry Count {} ", queueName, RetrySynchronizationManager.*getContext*().getRetryCount());  
 int retryCount= RetrySynchronizationManager.*getContext*().getRetryCount();  
 String status ="";  
 try  
 {  
  
 if(qMgr ==null)  
 {  
 *log*.info("MQ Queue Manager is null so creating new Queue Manager");  
 String mqManagerUniqueKey = mqConfiguration.getRemoteAddress() + mqConfiguration.getChannelName() + mqConfiguration.getQueueManagerName();  
 synchronized (this)  
 {  
 *log*.info("Creating queue Manager for queue .... {}" , queueName);  
 qMgr = getMqManager(mqConfiguration, mqManagerUniqueKey,uniqueKey);  
 }  
 }  
 *log*.info("MQ Queue Manager is not null and details are {}" , qMgr);  
 MQQueue queue = qMgr.accessQueue(queueName, CMQC.*MQOO\_OUTPUT* | CMQC.*MQOO\_FAIL\_IF\_QUIESCING*);  
  
 MQPutMessageOptions pmo = new MQPutMessageOptions();  
 pmo.options = CMQC.*MQPMO\_SYNCPOINT*;  
 // create message  
 MQMessage message = new MQMessage();  
 message.persistence = CMQC.*MQPER\_PERSISTENT*;  
 message.format = CMQC.*MQFMT\_STRING*;  
  
 message.writeString(messageToSend);  
 queue.put(message, pmo);  
 *log*.info("Message has been pushed to queue");  
 queue.close();  
 qMgr.commit();  
 updateOutgoingMqMessage(outgoingMqMessage, (long) retryCount,"", MqConstants.*MQ\_SUCCESSFUL\_STATUS*).subscribe();  
 status= MqConstants.*MQ\_SUCCESSFUL\_STATUS*;  
 }  
 catch ( Throwable e)  
 {  
 *log*.error("Exception in sendMessageToQueue on Retry count {} and message{}" , RetrySynchronizationManager.*getContext*().getRetryCount() , e.getMessage());  
 if (null != qMgr)  
 {  
 try  
 {  
 qMgr.disconnect();  
 }  
 catch (MQException mqExp)  
 {  
 *log*.error("Failed to disconnect queue manager {}", mqExp.getMessage());  
 }  
 }  
 updateOutgoingMqMessage(outgoingMqMessage, (long) retryCount,e.getMessage(), MqConstants.*MQ\_UN\_SUCCESSFUL\_STATUS*).subscribe();  
 throw new MqConnectionException("Mq connection failed || " + e.getMessage(), e);  
 }  
 *log*.debug("Exiting from sendMessageToQueue");  
 return Mono.*just*(status);  
}  
  
*/\*\*  
 \* Get Mq Manager  
 \* @param mqConfiguration mqConfiguration object  
 \* @param mqManagerUniqueKey mqManagerUniqueKey  
 \* @return MQQueueManager  
 \* @throws MQException exception  
 \*/*public MQQueueManager getMqManager(MqConfiguration mqConfiguration, String mqManagerUniqueKey, String uniqueKey) throws MQException  
{  
 *log*.info("create new queueManager in SendMessageMqUtil");  
 MQQueueManager mqManager = null;  
  
 Hashtable<String, Object> props = *getStringObjectHashtable*(mqConfiguration);  
  
 *log*.info("MQManager properties added in SendMessageMqUtil {}.." , props.toString());  
  
 mqManager = new MQQueueManager(mqConfiguration.getQueueManagerName(), props);  
  
 if(mqConnectionsInMemory.getMqManagerMap().get(uniqueKey) == null)  
 {  
 mqConnectionsInMemory.getMqManagerMap().put(uniqueKey,mqManager);  
 mqConnectionsInMemory.getMqConfigurationConcurrentHashMap().put(uniqueKey,mqConfiguration);  
 }  
  
 *log*.info("New Queue manager created in SendMessageMqUtil {}.." , mqManager.toString());  
 Map<String, MQQueueManager> managerMap = mqConnectionsInMemory.getMqManagerMap();  
 managerMap.put(mqManagerUniqueKey, mqManager);  
  
 return mqManager;  
}

*/\*\*  
\*  
\* © 2007-2022 Fidelity National Information Services, Inc.  
\* and/or its subsidiaries - All Rights Reserved worldwide.  
\* This document is protected under the trade secret and copyright laws as the  
\* property of Fidelity National Information Services, Inc. and/or its subsidiaries.  
\* Copying, reproduction or distribution should be limited and only to  
\* employees with a "need to know" to do their job.  
\* Any disclosure of this document to third parties is strictly prohibited.  
\*/*package com.fis.d1.account.jwt;  
  
import java.io.IOException;  
import java.security.GeneralSecurityException;  
import java.security.Key;  
import java.security.KeyFactory;  
import java.security.KeyStore;  
import java.security.PrivateKey;  
import java.security.spec.PKCS8EncodedKeySpec;  
import java.time.Instant;  
import java.util.Base64;  
import java.util.Date;  
import java.util.UUID;  
  
import jakarta.annotation.PostConstruct;  
  
import org.apache.commons.lang3.StringUtils;  
import org.slf4j.Logger;  
import org.slf4j.LoggerFactory;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.core.io.Resource;  
import org.springframework.core.io.ResourceLoader;  
import org.springframework.stereotype.Component;  
  
import com.fis.d1.account.util.ConfigurationManager;  
import com.fis.bpf.spring.error.fisapi.exception.ExceptionBuilder;  
  
import io.jsonwebtoken.Jwts;  
  
*/\*\*  
 \* @author Heera Sridhar  
 \*  
 \*/*@Component  
public class JwtTokenGenerator {  
 private static final Logger *LOGGER* = LoggerFactory.*getLogger*(JwtTokenGenerator.class);  
   
 @Autowired  
 private ConfigurationManager configManager;  
   
 @Autowired  
 private ResourceLoader resourceLoader;  
   
 private PrivateKey privateKey = null;  
   
 @PostConstruct  
 protected void loadPrivateKey() throws IOException, GeneralSecurityException {  
 privateKey = getPrivateKey();  
 }  
   
 public String createJwtSigned() {  
 *LOGGER*.debug("Generating JWT Token");  
   
 if(privateKey == null) {  
 throw ExceptionBuilder.*technical*().message("JWT private key not found").detail()  
 .code("INT-80001").message("Internal Server Error").add().build();  
 }  
   
 Instant now = Instant.*now*();  
 Date exp = new Date(new Date().getTime() + (1000L\*60L\*30L)); //1000ms \* 60s \* 30m  
 *LOGGER*.debug("Expiration Time {}", exp);  
 String jwtToken = Jwts.*builder*()  
 .setHeaderParam("typ","JWT")  
 .setAudience(configManager.getConfigStr("aud"))  
 .setIssuer(configManager.getConfigStr("iss"))  
 .setId(randomHex())  
 .setNotBefore(Date.*from*(now))  
 .setSubject(configManager.getConfigStr("sub"))  
 .setExpiration(exp)  
 .setIssuedAt(Date.*from*(now))  
 .claim("PLATFORM\_APPS", configManager.getConfigStr("platform\_apps"))  
 .claim("UserID", configManager.getConfigStr("sub"))  
 .claim("FIID", configManager.getConfigStr("fiid"))  
 .claim("accessToken", randomHex())  
 .claim("expires\_in", exp)  
 .signWith(privateKey)  
 .compact();  
  
 *LOGGER*.debug("JWT Token Generated value: {}", jwtToken);  
 return jwtToken;  
 }  
   
 private String randomHex() {  
 *LOGGER*.debug("Generating random Hex");  
 String randomVal= UUID.*randomUUID*().toString().replace("-", "");  
 *LOGGER*.debug("RandomHex Value {}", randomVal);  
 return randomVal;  
 }  
  
 private PrivateKey getPrivateKey() throws IOException, GeneralSecurityException {  
 PrivateKey pk = getPrivateKeyFromJKS();  
 if(pk == null) {  
 pk = getPrivateKeyFromConfigString();  
 }  
 return pk;  
 }  
   
 private PrivateKey getPrivateKeyFromConfigString() throws GeneralSecurityException {  
 *LOGGER*.info("Creating JWT private key via config string");  
 String rsaPrivateKey = configManager.getUnscopedConfigStr("jwt\_pkcs\_file", false);  
   
 if(StringUtils.*isNotBlank*(rsaPrivateKey)) {  
 rsaPrivateKey = rsaPrivateKey.replace("-----BEGIN PRIVATE KEY-----", "");  
 rsaPrivateKey = rsaPrivateKey.replace("-----END PRIVATE KEY-----", "");  
 PKCS8EncodedKeySpec keySpec = new PKCS8EncodedKeySpec(Base64.*getDecoder*().decode(rsaPrivateKey));  
 KeyFactory kf = KeyFactory.*getInstance*("RSA");  
 PrivateKey privKey = kf.generatePrivate(keySpec);  
 *LOGGER*.info("JWT private key created via config string");  
 return privKey;  
 } else {  
 *LOGGER*.info("JWT private key config string NOT FOUND - private key was not created.");  
 return null;  
 }  
 }  
   
 private PrivateKey getPrivateKeyFromJKS() throws IOException, GeneralSecurityException {  
 *LOGGER*.info("Creating JWT private key via JKS");  
 String ksFileName = configManager.getUnscopedConfigStr("jwt\_keystore\_file", false);  
   
 if(StringUtils.*isNotBlank*(ksFileName)) {  
 String ksCred = configManager.getUnscopedConfigStr("jwt\_keystore\_pwd", true);  
 String certAlias = configManager.getUnscopedConfigStr("jwt\_cert\_alias", true);  
   
 Resource resource = resourceLoader.getResource(ksFileName);  
 KeyStore jksKeystore = KeyStore.*getInstance*("JKS");  
 jksKeystore.load(resource.getInputStream(), ksCred.toCharArray());  
 Key key = jksKeystore.getKey(certAlias, ksCred.toCharArray());  
 PKCS8EncodedKeySpec keySpec = new PKCS8EncodedKeySpec(key.getEncoded());  
 KeyFactory kf = KeyFactory.*getInstance*("RSA");  
 PrivateKey privKey = kf.generatePrivate(keySpec);  
 *LOGGER*.info("JWT private key created via JKS");  
   
 return privKey;  
 }else {  
 *LOGGER*.info("JKS Keystore NOT FOUND for JWT private key - it was not created.");  
 return null;  
 }  
 }  
}

---------------------------

StartUp

package com.fis.bps.pmt.sys.startup;  
  
  
import com.fis.bps.base.common.utils.AppUtils;  
  
import com.fis.bps.base.exception.models.RuntimeExceptionClass;  
import com.fis.bps.base.exception.models.ServerErrorException;  
import com.fis.bps.base.mq.model.MqConfigInfo;  
import com.fis.bps.base.mq.model.MqConfiguration;  
import com.fis.bps.base.mq.model.MqConnectionDetails;  
import com.fis.bps.base.mq.repository.MqConfigInfoRepository;  
import com.fis.bps.base.mq.repository.MqConnectionDetailsRepository;  
import com.fis.bps.pmt.sys.listner.AckListener;  
import lombok.AllArgsConstructor;  
import lombok.extern.slf4j.Slf4j;  
  
import org.springframework.boot.context.event.ApplicationReadyEvent;  
import org.springframework.context.event.EventListener;  
import org.springframework.stereotype.Component;  
import reactor.core.publisher.Flux;  
import reactor.core.publisher.Mono;  
  
import javax.jms.JMSException;  
import java.nio.charset.StandardCharsets;  
import java.util.List;  
import java.util.ListIterator;  
  
@Component  
@Slf4j  
@AllArgsConstructor  
*/\*\*  
 \* This Class is basically for sending failed message  
 \* During the container or pod start up  
 \* we are sending both incoming and acknack failed  
 \* message  
 \*  
 \*/*public class MqListenerStartupEvent {  
  
 MqConfigInfoRepository mqConfigInfoRepository;  
 MqConnectionDetailsRepository mqConnectionDetailsRepository;  
 AppUtils appUtils;  
 AckListener ackListener;  
  
 @EventListener(ApplicationReadyEvent.class)  
 public Flux<Void> getAllMetaConfigData()  
 {  
 *log*.info("Launching MqListener Startup");  
  
 return startApiConfigMqListeners();  
 }  
  
 public Flux<Void> startApiConfigMqListeners()  
 {  
 *log*.info("Entered into start Api Config Mq Listeners method");  
  
 return mqConfigInfoRepository.findByProviderCode("OPF").flatMap(this::convertToMqConfiguration);  
  
 }  
  
 public Mono<Void> convertToMqConfiguration(MqConfigInfo mqConfigInfo)  
 {  
 *log*.info("Creating Mq Configuration object");  
 Mono<MqConnectionDetails> connectionDetailsFlux = mqConnectionDetailsRepository.findByMqConfigId(mqConfigInfo.getId());  
 return connectionDetailsFlux.flatMap(connectionDetails -> {  
  
 String eventData = new String(connectionDetails.getPropertyDetails(), StandardCharsets.*UTF\_8*);  
 MqConfiguration mqConfiguration = appUtils.convertToObject(eventData, MqConfiguration.class);  
 *log*.info("MqConfiguration Object {} " , mqConfiguration.toString());  
 try {  
 ackListener.ackMessageMqListener(mqConfiguration);  
 } catch (JMSException e) {  
 *log*.error("MqListenerStartupEvent.convertToMqConfiguration error {}", e.getMessage());  
 throw new RuntimeExceptionClass(e.getMessage());  
 }  
  
 return Mono.*empty*();  
 });  
  
  
 }  
  
  
}

package com.fis.bps.pmt.sys.listner;  
  
import com.fis.bps.base.common.utils.AppUtils;  
import com.fis.bps.base.mq.consumer.MqConsumerService;  
import com.fis.bps.base.mq.exception.JmsConnectionExceptionListener;  
import com.fis.bps.base.mq.exception.MqConnectionException;  
import com.fis.bps.base.mq.model.MqConfiguration;  
import com.fis.bps.base.mq.repository.PaymentInfoRepository;  
import com.fis.bps.base.mq.repository.PaymentTransactionEntityRepository;  
import com.fis.bps.base.mq.util.IbmMqUtil;  
import com.fis.bps.base.mq.util.MqConnectionsInMemory;  
import com.fis.bps.kafka.producer.KafkaMessageProducer;  
import com.ibm.mq.jms.MQQueueConnectionFactory;  
import lombok.AllArgsConstructor;  
import lombok.extern.slf4j.Slf4j;  
import org.springframework.beans.factory.annotation.Value;  
import org.springframework.stereotype.Component;  
  
  
import javax.jms.ExceptionListener;  
import javax.jms.JMSException;  
import javax.jms.MessageListener;  
  
@Component  
@Slf4j  
@AllArgsConstructor  
public class AckListener {  
  
  
 AppUtils appUtils;  
 MqConnectionsInMemory mqConnectionsInMemory;  
 IbmMqUtil ibmMqUtil;  
 MqConsumerService mqConsumerService;  
  
 PaymentInfoRepository paymentInfoRepository;  
 PaymentTransactionEntityRepository paymentTransactionEntityRepository;  
  
 KafkaMessageProducer kafkaMessageProducer;  
  
  
  
 public void ackMessageMqListener(MqConfiguration mqConfiguration) throws MqConnectionException, JMSException {  
  
 *log*.info("Ack Nak Message Mq Listener with Configuration Details {}", appUtils.getRequestToJson(mqConfiguration));  
 String mqListenerQueueUniqueKey = mqConfiguration.getRemoteAddress() + ":" + mqConfiguration.getQueueManagerName() + ":" + mqConfiguration.getAckNAckQueueName();  
 if (!mqConnectionsInMemory.getMqListenerStartedMap().containsKey(mqListenerQueueUniqueKey) ||  
 Boolean.*TRUE*.equals(!mqConnectionsInMemory.getMqListenerStartedMap().get(mqListenerQueueUniqueKey))) {  
  
 MQQueueConnectionFactory mqQueueConnectionFactory = ibmMqUtil.getMqQueueConnectionFactory(mqConfiguration);  
 MessageListener ackNackMessageListener= new OpfAcKNAckMessageListener(mqConfiguration.getAckNAckQueueName(),  
 appUtils,paymentInfoRepository,paymentTransactionEntityRepository,mqConfiguration.getTopicNameAckNack(),  
 kafkaMessageProducer);  
  
 ExceptionListener exceptionListener = new JmsConnectionExceptionListener(mqListenerQueueUniqueKey, mqConnectionsInMemory);  
 mqConsumerService.createMqConsumer(mqQueueConnectionFactory, mqConfiguration.getAckNAckQueueName(),ackNackMessageListener,  
 exceptionListener, mqConfiguration.getUserIdProperty());  
 mqConnectionsInMemory.getMqListenerStartedMap().put(mqListenerQueueUniqueKey, true);  
  
 }  
  
 }  
  
}

package com.fis.bps.pmt.sys.listner;  
  
import com.fis.bpf.spring.error.fisapi.exception.ExceptionBuilder;  
import com.fis.bps.base.common.utils.AppUtils;  
import com.fis.bps.base.mq.model.PaymentInfoModel;  
import com.fis.bps.base.mq.model.PaymentTransactionEntity;  
import com.fis.bps.base.mq.repository.PaymentInfoRepository;  
import com.fis.bps.base.mq.repository.PaymentTransactionEntityRepository;  
import com.fis.bps.kafka.producer.KafkaMessageProducer;  
import com.fis.bps.omni.base.pain.model.transactions.ApprovalHistoryModel;  
import com.fis.bps.omni.base.pain.model.transactions.EnrichedDataModel;  
import com.fis.bps.pmt.sys.constants.SystemConstants;  
import com.jayway.jsonpath.JsonPath;  
import lombok.AllArgsConstructor;  
import lombok.extern.slf4j.Slf4j;  
import org.apache.commons.text.StringEscapeUtils;  
import org.json.JSONObject;  
import org.json.XML;  
import org.springframework.beans.factory.annotation.Value;  
import org.springframework.stereotype.Component;  
import reactor.core.publisher.Mono;  
  
import javax.jms.JMSException;  
import javax.jms.Message;  
import javax.jms.MessageListener;  
import javax.jms.TextMessage;  
import java.nio.charset.StandardCharsets;  
import java.time.LocalDateTime;  
import java.time.ZoneId;  
import java.util.ArrayList;  
import java.util.List;  
  
  
@Slf4j  
@AllArgsConstructor  
public class OpfAcKNAckMessageListener implements MessageListener {  
  
 String ackNAckQueueName;  
 AppUtils appUtils;  
 PaymentInfoRepository paymentInfoRepository;  
 PaymentTransactionEntityRepository paymentTransactionEntityRepository;  
 String ackNackTopicName;  
 KafkaMessageProducer kafkaMessageProducer;  
  
 @Override  
 public void onMessage(Message message) {  
 try {  
  
 *log*.info("OPF Ack NAck - Message received from Queue .. {} " , ackNAckQueueName);  
 String receivedMsg = ((TextMessage) message).getText();  
 parseApiInboundMessage(receivedMsg).subscribe();  
 *log*.info("parse Api Inbound Message method finish");  
 } catch (JMSException e) {  
 *log*.error(e.getMessage(), e);  
 }  
  
 }  
  
 */\*\*  
 \* Method to prse the incoming message and perform Some Operations on it  
 \* @param receivedMsg String receivedMsg  
 \* @return  
 \*/* public Mono<Void> parseApiInboundMessage(String receivedMsg) {  
 *log*.info("Entered into parse opf incoming message for ack NAck received {} " , StringEscapeUtils.*escapeJava*(receivedMsg));  
  
 String originalMessage =receivedMsg;  
  
 receivedMsg = receivedMsg.replaceAll("(<\\?[^<]\*\\?>)?", ""). /\* remove preamble \*/  
 replaceAll("xmlns.\*?(\"|\').\*?(\"|\')", "") /\* remove xmlns declaration \*/  
 .replaceAll("(<)(\\w+:)(.\*?>)", "$1$3") /\* remove opening tag prefix \*/  
 .replaceAll("(</)(\\w+:)(.\*?>)", "$1$3"); /\* remove closing tags prefix \*/  
 JSONObject xmlJSONObj = XML.*toJSONObject*(receivedMsg);  
 String jsonMessage = xmlJSONObj.toString(4);  
  
 *log*.info(" opf incoming Ack Nack message json format {}",jsonMessage);  
  
  
 String uetrId=JsonPath.*read*(jsonMessage, "$.Document.CstmrPmtStsRpt.OrgnlPmtInfAndSts.TxInfAndSts.OrgnlUETR");  
 String isoStatus = JsonPath.*read*(jsonMessage, "$.Document.CstmrPmtStsRpt.OrgnlPmtInfAndSts.TxInfAndSts.TxSts");  
 String transactionStatus =getTransactionStatus(isoStatus);  
  
 *log*.info("ACK NACK Original uetr id {} , with iso status {} and transaction status {} is",uetrId,isoStatus,transactionStatus);  
 return updateTransaction(SystemConstants.*USER\_NAME\_CONSTANT*,uetrId,isoStatus,transactionStatus).flatMap(b-> {  
 kafkaMessageProducer.send(ackNackTopicName,originalMessage);  
 return Mono.*empty*();  
 });  
  
  
 }  
  
  
 */\*\*  
 \* Update Transaction data method  
 \* @param user : user id  
 \* @param uetr : Uetr ID  
 \* @param status : ISO status  
 \* @param transactionStatus : Transaction Status  
 \* @return Mono of PaymentInfoModel  
 \*/* public Mono<PaymentInfoModel> updateTransaction(String user, String uetr,String status, String transactionStatus)  
 {  
 return paymentTransactionEntityRepository.existsPaymentTransactionEntityByUetr(uetr).flatMap(isExists->{  
 if(Boolean.*TRUE*.equals(isExists))  
 {  
  
 return paymentTransactionApproved(user,uetr,status,transactionStatus)  
 .flatMap(paymentTransactionModelData -> paymentInfoUpdate(user,paymentTransactionModelData.getPaymentInfoId(),status,transactionStatus));  
 }else {  
 *log*.info("No Data exists with Uetr {} in database" , uetr);  
 return Mono.*just*( new PaymentInfoModel());  
 }  
 });  
  
 }  
  
 */\*\*  
 \* Update Transaction Details  
 \* @param user : user id  
 \* @param uter : Uetr ID  
 \* @param status : ISO status  
 \* @param transactionStatus : Transaction Status  
 \* @return Mono of PaymentTransactionEntity  
 \*/* public Mono<PaymentTransactionEntity> paymentTransactionApproved(String user,String uter, String status, String transactionStatus)  
 {  
  
 *log*.info("Payment Transaction Approve method started for uetr {}" ,uter);  
 return paymentTransactionEntityRepository.findByUetr(uter).flatMap(paymentTransactionModel->{  
  
 EnrichedDataModel er =getEnrichedData(status,paymentTransactionModel.getTransactionDetails());  
 paymentTransactionModel.setTransactionDetails(appUtils.getRequestToJson(er).getBytes(StandardCharsets.*UTF\_8*));  
 paymentTransactionModel.setTransactionStatus(transactionStatus);  
 paymentTransactionModel.setUpdatedBy(user);  
 paymentTransactionModel.setUpdatedDate(LocalDateTime.*now*(ZoneId.*of*(SystemConstants.*ZONE\_ID\_AMERICA*)));  
 return paymentTransactionEntityRepository.save(paymentTransactionModel);  
 });  
  
 }  
  
  
 */\*\*  
 \* Update Transaction Details batch Data  
 \* @param userId : user id  
 \* @param id : Payment Info Model ID  
 \* @param status : ISO status  
 \* @param transactionStatus : Transaction Status  
 \* @return Mono of PaymentInfoModel  
 \*/* public Mono<PaymentInfoModel> paymentInfoUpdate(String userId, Long id, String status,String transactionStatus)  
 {  
  
 *log*.info("Payment info update method started for id {}" ,id);  
 return paymentInfoRepository.findById(id).flatMap(paymentInfoModel->{  
  
 EnrichedDataModel er =getEnrichedData(status,paymentInfoModel.getEnrichData());  
 paymentInfoModel.setEnrichData(appUtils.getRequestToJson(er).getBytes(StandardCharsets.*UTF\_8*));  
 paymentInfoModel.setStatus(transactionStatus);  
 paymentInfoModel.setUpdatedBy(userId);  
 paymentInfoModel.setUpdatedDate(LocalDateTime.*now*(ZoneId.*of*(SystemConstants.*ZONE\_ID\_AMERICA*)));  
 return paymentInfoRepository.save(paymentInfoModel);  
  
 });  
  
 }  
  
 */\*\*  
 \* Get Enrich Data After Updating iso status and transaction status  
 \* @param isoStatus : isoStatus  
 \* @param enrichData : byte[] enrichData  
 \* @return EnrichedDataModel  
 \*/* public EnrichedDataModel getEnrichedData(String isoStatus, byte[] enrichData)  
 {  
  
 *log*.info("Setting of iso Status started ");  
  
 String data = new String(enrichData, StandardCharsets.*UTF\_8*);  
 EnrichedDataModel enrichedDataModel = appUtils.convertToObject(data, EnrichedDataModel.class);  
  
  
 String finalStatus = getTransactionStatus(isoStatus);  
 enrichedDataModel.getMessageHeader().getPaymentInformation().forEach(paymentInformation -> {  
  
 paymentInformation.getPaymentInfoStatus().setSts(finalStatus);  
 paymentInformation.getPaymentInfoStatus().setIsoSts(isoStatus);  
 paymentInformation.getCreditTransferTransactionInformationList().forEach(creditTransferTransactionInformation -> {  
  
 creditTransferTransactionInformation.getTransactionStatus().setStatus(finalStatus);  
 creditTransferTransactionInformation.getTransactionStatus().setIsoStatus(isoStatus);  
 });  
 });  
  
 *log*.info("Setting of iso Status finished ");  
 return enrichedDataModel;  
 }  
  
  
 */\*\*  
 \* Get Transaction status based on iso status  
 \* @param isoStatus : iso status  
 \* @return string  
 \*/* public String getTransactionStatus(String isoStatus)  
 {  
 String status = SystemConstants.*TRANSACTION\_STATUS\_CONFIRMED*;  
  
 if(SystemConstants.*ISO\_STATUS\_ACSC\_CONSTANT*.equalsIgnoreCase(isoStatus))  
 {  
 status =SystemConstants.*TRANSACTION\_STATUS\_PROCESSED*;  
 }  
 if(SystemConstants.*ISO\_STATUS\_RJCT\_CONSTANT*.equalsIgnoreCase(isoStatus))  
 {  
 status =SystemConstants.*TRANSACTION\_STATUS\_REJECTED*;  
 }  
  
 return status;  
 }  
  
}

package com.fis.bps.pmt.fednow.repository;  
  
import com.fis.bps.pmt.fednow.model.entity.PaymentTransactionModel;  
import lombok.extern.slf4j.Slf4j;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.data.r2dbc.core.R2dbcEntityTemplate;  
import org.springframework.r2dbc.core.DatabaseClient;  
import org.springframework.stereotype.Repository;  
import reactor.core.publisher.Flux;  
import reactor.core.publisher.Mono;  
  
import java.time.LocalDate;  
import java.time.ZoneId;  
import java.util.Date;  
import java.util.List;  
  
@Repository  
@Slf4j  
public class GetFednowTransactionsRespository {  
  
 @Autowired  
 private R2dbcEntityTemplate entityTemplate;  
  
 */\*\*  
 \* Method To Get Payment Transaction Data for Pending Transactions  
 \* @return Flux*<*PaymentTransactionModel*>  
 *\*/* public Flux<PaymentTransactionModel> findAllByCustomerIdAndStatus(String orgId, String channel, String customerId, String status, Integer pageNumber, Integer pageSize, String sortField, boolean ascending) {  
 String baseQuery = "WITH TotalCount AS (" +  
 " SELECT COUNT(\*) AS totalCount " +  
 " FROM PAYMENT\_TRANSACTION PT JOIN PAYMENT\_INFO PI ON PT.PAYMENT\_INFO\_ID = PI.ID " +  
 " WHERE PI.ORGANIZATION\_ID = :orgId AND PI.CHANNEL = :channel AND PT.CUSTOMER\_ID = :customerId AND UPPER(PT.STATUS) = UPPER(:status) " +  
 " ) " +  
 " SELECT PT.\*, TC.totalCount, " +  
 " CASE WHEN TC.totalCount > 0 THEN 'EXISTS' ELSE 'NOT EXISTS' END AS dataExists " +  
 " FROM PAYMENT\_TRANSACTION PT " +  
 " JOIN PAYMENT\_INFO PI ON PT.PAYMENT\_INFO\_ID = PI.ID " +  
 " JOIN TotalCount TC ON 1=1 " +  
 " WHERE PI.ORGANIZATION\_ID = :orgId AND PI.CHANNEL = :channel AND PT.CUSTOMER\_ID = :customerId AND UPPER(PT.STATUS) = UPPER(:status) ";  
  
 String orderByClause = "ORDER BY " + sortField + (ascending ? " ASC" : " DESC");  
  
 int offset = pageNumber \* pageSize;  
  
 String offsetClause = " OFFSET "+ offset +" ROWS FETCH NEXT "+pageSize + " ROWS ONLY ";  
  
 String finalQuery = baseQuery + orderByClause + offsetClause;  
  
 *log*.info("Order and Offset Clause :: {} - {} ",orderByClause,offsetClause);  
  
 DatabaseClient.GenericExecuteSpec executeSpec = entityTemplate.getDatabaseClient().sql(finalQuery)  
 .bind("orgId", orgId)  
 .bind("channel", channel)  
 .bind("customerId", customerId)  
 .bind("status", status);  
  
 return executeSpec.map((row, rowMetadata) -> {  
 PaymentTransactionModel transaction = entityTemplate.getConverter().read(PaymentTransactionModel.class, row);  
 Long totalCount = row.get("totalCount", Long.class);  
 transaction.setTotalCount(totalCount);  
 return transaction;  
 }).all();  
  
 }  
  
 */\*\*  
 \* Method To Get Payment Transaction Data for Completed Transactions  
 \* @return Flux*<*PaymentTransactionModel*>  
 *\*/* public Flux<PaymentTransactionModel> findAllByCustomerIdAndStatusAndAccountId(String orgId, String channel, String customerId, List<String> statusList,  
 List<String> accounts, String fromDate, String toDate, Integer pageNumber, Integer pageSize,  
 String sortColumn, boolean ascending) {  
 String baseQuery = "WITH TotalCount AS (" +  
 " SELECT COUNT(\*) AS totalCount " +  
 " FROM PAYMENT\_TRANSACTION PT JOIN PAYMENT\_INFO PI ON PT.PAYMENT\_INFO\_ID = PI.ID " +  
 " WHERE PI.ORGANIZATION\_ID = :orgId AND PI.CHANNEL = :channel AND UPPER(PT.CUSTOMER\_ID) = UPPER(:customerId) " +  
 " AND PT.ACCOUNT\_ID IN (:accounts) " +  
 " AND PT.CREATION\_DATE BETWEEN TO\_TIMESTAMP(:fromDate, 'YYYY-MM-DD\"T\"HH24:MI:SS.FF3') " +  
 " AND TO\_TIMESTAMP(:toDate, 'YYYY-MM-DD\"T\"HH24:MI:SS.FF3') ";  
  
 if (statusList.size() == 1) {  
 baseQuery += " AND UPPER(PT.STATUS) = :status ";  
 } else {  
 baseQuery += " AND UPPER(PT.STATUS) IN (:statusList) ";  
 }  
  
 baseQuery += " ) " +  
 " SELECT PT.\*, TC.totalCount, " +  
 " CASE WHEN TC.totalCount > 0 THEN 'EXISTS' ELSE 'NOT EXISTS' END AS dataExists " +  
 " FROM PAYMENT\_TRANSACTION PT " +  
 " JOIN PAYMENT\_INFO PI ON PT.PAYMENT\_INFO\_ID = PI.ID " +  
 " JOIN TotalCount TC ON 1=1 " +  
 " WHERE PI.ORGANIZATION\_ID = :orgId AND PI.CHANNEL = :channel AND UPPER(PT.CUSTOMER\_ID) = UPPER(:customerId) " +  
 " AND PT.ACCOUNT\_ID IN (:accounts) " +  
 " AND PT.CREATION\_DATE BETWEEN TO\_TIMESTAMP(:fromDate, 'YYYY-MM-DD\"T\"HH24:MI:SS.FF3') " +  
 " AND TO\_TIMESTAMP(:toDate, 'YYYY-MM-DD\"T\"HH24:MI:SS.FF3') ";  
  
 if (statusList.size() == 1) {  
 baseQuery += " AND UPPER(PT.STATUS) = :status ";  
 } else {  
 baseQuery += " AND UPPER(PT.STATUS) IN (:statusList) ";  
 }  
  
 String orderByClause = " ORDER BY " + sortColumn + (ascending ? " ASC" : " DESC");  
  
 int offset = pageNumber \* pageSize;  
  
 String offsetClause = " OFFSET "+ offset +" ROWS FETCH NEXT "+pageSize + " ROWS ONLY ";  
  
 String finalQuery = baseQuery + orderByClause + offsetClause;  
  
 *log*.info("Order and Offset Clause :: {} - {} ",orderByClause,offsetClause);  
  
 DatabaseClient.GenericExecuteSpec executeSpec = entityTemplate.getDatabaseClient().sql(finalQuery)  
 .bind("orgId", orgId)  
 .bind("channel", channel)  
 .bind("customerId", customerId)  
 .bind("accounts", accounts)  
 .bind("fromDate", fromDate)  
 .bind("toDate", toDate);  
  
 if (statusList.size() == 1) {  
 executeSpec = executeSpec.bind("status", statusList.get(0));  
 } else {  
 executeSpec = executeSpec.bind("statusList", statusList);  
 }  
  
 return executeSpec.map((row, rowMetadata) -> {  
 PaymentTransactionModel transaction = entityTemplate.getConverter().read(PaymentTransactionModel.class, row);  
 Long totalCount = row.get("totalCount", Long.class);  
 transaction.setTotalCount(totalCount);  
 return transaction;  
 }).all();  
 }  
  
 */\*\*  
 \* Method To Check if Valid Status for Completed Transaction  
 \* @return Mono*<*Long*>  
 *\*/* public Mono<Long> countPaymentTransactionModelByCustomerIdAndAccountIdAndStatus(String customerId, List<String> accounts, List<String> statuses) {  
 String baseQuery = "SELECT COUNT(DISTINCT PT.ACCOUNT\_ID) " +  
 " FROM PAYMENT\_TRANSACTION PT " +  
 " JOIN PAYMENT\_INFO PI ON PT.PAYMENT\_INFO\_ID = PI.ID " +  
 " WHERE UPPER(PT.CUSTOMER\_ID) = UPPER(:customerId) " +  
 " AND PT.ACCOUNT\_ID IN (:accounts) ";  
  
 *log*.info("Count Query : Base Query :: {}",baseQuery);  
 if (statuses.size() == 1) {  
 baseQuery += " AND UPPER(PT.STATUS) = :status";  
 } else {  
 baseQuery += " AND UPPER(PT.STATUS) IN (:statuses)";  
 }  
  
 *log*.info("Count Query : Final Query :: {}",baseQuery);  
 DatabaseClient.GenericExecuteSpec executeSpec = entityTemplate.getDatabaseClient().sql(baseQuery)  
 .bind("customerId", customerId)  
 .bind("accounts", accounts);  
  
 if (statuses.size() == 1) {  
 executeSpec = executeSpec.bind("status", statuses.get(0));  
 } else {  
 executeSpec = executeSpec.bind("statuses", statuses);  
 }  
 *log*.info("Bound Parameters - customerId: {}, accounts: {}, statuses: {} ", customerId, accounts, statuses);  
 return executeSpec.map((row, rowMetadata) -> {  
 Long count = row.get(0, Long.class);  
 *log*.info("Count Result: {}", count);  
 return count;  
 }).one();  
 }  
}

s