```java
package com.mybank.transactionservice;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Isolation;
import org.springframework.transaction.annotation.Transactional;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.math.BigDecimal;
import java.math.RoundingMode;
import java.time.LocalDateTime;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;
import java.util.Map;
import java.util.Objects;
import java.util.Optional;
import java.util.Set;
import java.util.UUID;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.ConcurrentSkipListSet;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicLong;
import java.util.function.Function;
import java.util.stream.Collectors;
import java.util.stream.Stream;

// region Global Constants and Configuration (Simulated)
class GlobalConfig {
    public static final String DEFAULT_CURRENCY = "USD";
    public static final int CURRENCY_SCALE = 2; // For BigDecimal precision
    public static final RoundingMode CURRENCY_ROUNDING_MODE = RoundingMode.HALF_EVEN;
    public static final BigDecimal MIN_ACCOUNT_BALANCE = new BigDecimal("0.00");
    public static final BigDecimal FRAUD_THRESHOLD_AMOUNT = new BigDecimal("10000.00");
    public static final int MAX_DAILY_TRANSACTIONS = 50;
    public static final long SCHEDULED_TASK_INTERVAL_MINUTES = 1;
    public static final int MAX_RETRIES = 3;
    public static final long RETRY_DELAY_SECONDS = 5;
    public static final int RATE_LIMIT_PER_MINUTE = 100; // Requests per minute per client/IP
    public static final long IDEMPOTENCY_KEY_EXPIRATION_MINUTES = 60; // How long to remember idempotency keys
}
// endregion

// region DTOs, Entities, Value Objects, Enums (Expanded for Real-World Complexity)

/**
 * Represents the status of a financial transaction.
 */
enum TransactionStatus {
    PENDING, COMPLETED, FAILED, REVERSED, REFUNDED, CANCELED, AUTHORIZED, CAPTURED, SETTLED
}

/**
 * Represents the type of a financial transaction.
 */
enum TransactionType {
    DEPOSIT, WITHDRAWAL, TRANSFER, PAYMENT, REFUND, REVERSAL, FEE, INTEREST, FX_CONVERSION, AUTHORIZATION, CAPTURE
}

/**
 * Represents the channel through which a transaction was initiated.
 */
enum TransactionChannel {
    WEB_PORTAL, MOBILE_APP, ATM, POS, API, BATCH, ADMIN
}

/**
 * Represents the type of account.
 */
enum AccountType {
    SAVINGS, CHECKING, CREDIT_CARD, LOAN, EXTERNAL
}

/**
 * Represents the status of an account.
 */
enum AccountStatus {
    ACTIVE, INACTIVE, CLOSED, SUSPENDED, BLOCKED
}

/**
 * Represents the status of a scheduled transaction.
 */
enum ScheduleStatus {
    PENDING, COMPLETED, FAILED, CANCELED, SKIPPED
}

/**
 * Represents a party involved in a transaction (e.g., source or destination).
 * This is a Value Object for better encapsulation.
 */
class AccountIdentifier {
    private Long accountId;
    private String accountNumber;
    private AccountType accountType; // e.g., INTERNAL, EXTERNAL

    public AccountIdentifier(Long accountId, String accountNumber, AccountType accountType) {
        this.accountId = accountId;
        this.accountNumber = accountNumber;
        this.accountType = accountType;
    }

    public Long getAccountId() { return accountId; }
    public String getAccountNumber() { return accountNumber; }
    public AccountType getAccountType() { return accountType; }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        AccountIdentifier that = (AccountIdentifier) o;
        return Objects.equals(accountId, that.accountId) &&
               Objects.equals(accountNumber, that.accountNumber) &&
               accountType == that.accountType;
    }

    @Override
    public int hashCode() {
        return Objects.hash(accountId, accountNumber, accountType);
    }

    @Override
    public String toString() {
        return "AccountIdentifier{" +
```

```
                "accountId=" + accountId +
                ", accountNumber='" + accountNumber + '\'' +
                ", accountType=" + accountType +
                '}';
    }
}

/**
 * Basic Account Holder Entity.
 */
class AccountHolder {
    private Long id;
    private String userId; // Unique identifier for the user/customer
    private String firstName;
    private String lastName;
    private String email;
    private String phoneNumber;
    private LocalDateTime registrationDate;
    private String KYCStatus; // e.g., "VERIFIED", "PENDING", "REJECTED"

    public AccountHolder() {}
    public AccountHolder(Long id, String userId, String firstName, String lastName, String email, String phoneNumber, String KYCStatus) {
        this.id = id;
        this.userId = userId;
        this.firstName = firstName;
        this.lastName = lastName;
        this.email = email;
        this.phoneNumber = phoneNumber;
        this.registrationDate = LocalDateTime.now();
        this.KYCStatus = KYCStatus;
    }

    // Getters
    public Long getId() { return id; }
    public String getUserId() { return userId; }
    public String getFirstName() { return firstName; }
    public String getLastName() { return lastName; }
    public String getEmail() { return email; }
    public String getPhoneNumber() { return phoneNumber; }
    public LocalDateTime getRegistrationDate() { return registrationDate; }
    public String getKYCStatus() { return KYCStatus; }

    // Setters
    public void setId(Long id) { this.id = id; }
    public void setUserId(String userId) { this.userId = userId; }
    public void setFirstName(String firstName) { this.firstName = firstName; }
    public void setLastName(String lastName) { this.lastName = lastName; }
    public void setEmail(String email) { this.email = email; }
    public void setPhoneNumber(String phoneNumber) { this.phoneNumber = phoneNumber; }
    public void setRegistrationDate(LocalDateTime registrationDate) { this.registrationDate = registrationDate; }
    public void setKYCStatus(String KYCStatus) { this.KYCStatus = KYCStatus; }
}


/**
 * Extended Account Entity.
 * Represents a bank account in the system, typically a database record.
 */
class Account {
    private Long id;
    private String accountNumber;
    private Long accountHolderId; // Link to AccountHolder
    private AccountType type; // e.g., SAVINGS, CHECKING
    private BigDecimal balance;
    private String currency;
    private AccountStatus status; // e.g., "ACTIVE", "INACTIVE", "CLOSED", "SUSPENDED"
    private BigDecimal minimumBalance; // Enforce minimum balance for certain account types
    private LocalDateTime creationDate;
    private LocalDateTime lastUpdateDate;
    private Long version; // For optimistic locking

    public Account() {}
    public Account(Long id, String accountNumber, Long accountHolderId, AccountType type, BigDecimal balance, String currency, AccountStatus status, BigDecimal minimumBalance) {
        this.id = id;
        this.accountNumber = accountNumber;
        this.accountHolderId = accountHolderId;
        this.type = type;
        this.balance = balance.setScale(GlobalConfig.CURRENCY_SCALE, GlobalConfig.CURRENCY_ROUNDING_MODE);
        this.currency = currency;
        this.status = status;
        this.minimumBalance = minimumBalance.setScale(GlobalConfig.CURRENCY_SCALE, GlobalConfig.CURRENCY_ROUNDING_MODE);
        this.creationDate = LocalDateTime.now();
        this.lastUpdateDate = LocalDateTime.now();
        this.version = 0L;
    }

    // Getters
    public Long getId() { return id; }
    public String getAccountNumber() { return accountNumber; }
    public Long getAccountHolderId() { return accountHolderId; }
    public AccountType getType() { return type; }
    public BigDecimal getBalance() { return balance; }
    public String getCurrency() { return currency; }
    public AccountStatus getStatus() { return status; }
    public BigDecimal getMinimumBalance() { return minimumBalance; }
    public LocalDateTime getCreationDate() { return creationDate; }
    public LocalDateTime getLastUpdateDate() { return lastUpdateDate; }
    public Long getVersion() { return version; }

    // Setters
    public void setId(Long id) { this.id = id; }
    public void setAccountNumber(String accountNumber) { this.accountNumber = accountNumber; }
    public void setAccountHolderId(Long accountHolderId) { this.accountHolderId = accountHolderId; }
    public void setType(AccountType type) { this.type = type; }
    public void setBalance(BigDecimal balance) {
        this.balance = balance.setScale(GlobalConfig.CURRENCY_SCALE, GlobalConfig.CURRENCY_ROUNDING_MODE);
        this.lastUpdateDate = LocalDateTime.now();
    }
    public void setCurrency(String currency) { this.currency = currency; }
    public void setStatus(AccountStatus status) { this.status = status; }
    public void setMinimumBalance(BigDecimal minimumBalance) { this.minimumBalance = minimumBalance.setScale(GlobalConfig.CURRENCY_SCALE, GlobalConfig.CURRENCY_ROUNDING_MODE); }
    public void setCreationDate(LocalDateTime creationDate) { this.creationDate = creationDate; }
    public void setLastUpdateDate(LocalDateTime lastUpdateDate) { this.lastUpdateDate = lastUpdateDate; }
    public void setVersion(Long version) { this.version = version; }

    public void incrementVersion() {
        if (this.version == null) {
            this.version = 0L;
        }
        this.version++;
    }
}

/**
 * Extended Transaction Entity.
 * Represents a single financial transaction record, typically a database record.
 */
class Transaction {
    private Long id;
```

```
    private String transactionId; // Client-provided or generated for idempotency
    private String systemTransactionId; // Internally generated unique ID
    private TransactionType type;
    private BigDecimal originalAmount; // Amount before any fees/FX conversion
    private String originalCurrency;
    private BigDecimal processedAmount; // Amount actually debited/credited after fees/FX
    private String processedCurrency;
    private BigDecimal exchangeRate; // If currency conversion occurred
    private BigDecimal fees; // Total fees applied to this transaction
    private AccountIdentifier sourceAccount; // Could be internal or external
    private AccountIdentifier destinationAccount; // Could be internal or external
    private String externalReferenceId; // Reference from external payment systems
    private String internalReferenceId; // For internal traceability (e.g., link to original transaction for reversal)
    private LocalDateTime timestamp;
    private LocalDateTime settlementDate; // When funds are actually settled
    private TransactionStatus status;
    private String description;
    private String metadata; // JSON or key-value for additional details
    private TransactionChannel channel; // E.g., WEB, MOBILE, API
    private Long initiatorUserId; // User who initiated the transaction
    private String approvalStatus; // E.g., PENDING_APPROVAL, APPROVED, REJECTED (for high-value tx)
    private Long parentTransactionId; // For refunds/reversals

    public Transaction() {}
    public Transaction(String transactionId, TransactionType type, BigDecimal originalAmount, String originalCurrency,
                       AccountIdentifier sourceAccount, AccountIdentifier destinationAccount, String description,
                       TransactionChannel channel, Long initiatorUserId) {
        this.transactionId = transactionId;
        this.systemTransactionId = UUID.randomUUID().toString(); // Always generate a system-wide unique ID
        this.type = type;
        this.originalAmount = originalAmount.setScale(GlobalConfig.CURRENCY_SCALE, GlobalConfig.CURRENCY_ROUNDING_MODE);
        this.originalCurrency = originalCurrency;
        this.processedAmount = originalAmount; // Initially same as original, will be adjusted by fees/FX
        this.processedCurrency = originalCurrency;
        this.sourceAccount = sourceAccount;
        this.destinationAccount = destinationAccount;
        this.timestamp = LocalDateTime.now();
        this.status = TransactionStatus.PENDING;
        this.description = description;
        this.channel = channel;
        this.initiatorUserId = initiatorUserId;
        this.approvalStatus = "N/A"; // Default
        this.fees = BigDecimal.ZERO.setScale(GlobalConfig.CURRENCY_SCALE, GlobalConfig.CURRENCY_ROUNDING_MODE);
    }

    // Getters
    public Long getId() { return id; }
    public String getTransactionId() { return transactionId; }
    public String getSystemTransactionId() { return systemTransactionId; }
    public TransactionType getType() { return type; }
    public BigDecimal getOriginalAmount() { return originalAmount; }
    public String getOriginalCurrency() { return originalCurrency; }
    public BigDecimal getProcessedAmount() { return processedAmount; }
    public String getProcessedCurrency() { return processedCurrency; }
    public BigDecimal getExchangeRate() { return exchangeRate; }
    public BigDecimal getFees() { return fees; }
    public AccountIdentifier getSourceAccount() { return sourceAccount; }
    public AccountIdentifier getDestinationAccount() { return destinationAccount; }
    public String getExternalReferenceId() { return externalReferenceId; }
    public String getInternalReferenceId() { return internalReferenceId; }
    public LocalDateTime getTimestamp() { return timestamp; }
    public LocalDateTime getSettlementDate() { return settlementDate; }
    public TransactionStatus getStatus() { return status; }
    public String getDescription() { return description; }
    public String getMetadata() { return metadata; }
    public TransactionChannel getChannel() { return channel; }
    public Long getInitiatorUserId() { return initiatorUserId; }
    public String getApprovalStatus() { return approvalStatus; }
    public Long getParentTransactionId() { return parentTransactionId; }

    // Setters
    public void setId(Long id) { this.id = id; }
    public void setTransactionId(String transactionId) { this.transactionId = transactionId; }
    public void setSystemTransactionId(String systemTransactionId) { this.systemTransactionId = systemTransactionId; }
    public void setType(TransactionType type) { this.type = type; }
    public void setOriginalAmount(BigDecimal originalAmount) { this.originalAmount = originalAmount.setScale(GlobalConfig.CURRENCY_SCALE, GlobalConfig.CURRENCY_ROUNDING_MODE); }
    public void setOriginalCurrency(String originalCurrency) { this.originalCurrency = originalCurrency; }
    public void setProcessedAmount(BigDecimal processedAmount) { this.processedAmount = processedAmount.setScale(GlobalConfig.CURRENCY_SCALE, GlobalConfig.CURRENCY_ROUNDING_MODE); }
    public void setProcessedCurrency(String processedCurrency) { this.processedCurrency = processedCurrency; }
    public void setExchangeRate(BigDecimal exchangeRate) { this.exchangeRate = exchangeRate; }
    public void setFees(BigDecimal fees) { this.fees = fees.setScale(GlobalConfig.CURRENCY_SCALE, GlobalConfig.CURRENCY_ROUNDING_MODE); }
    public void setSourceAccount(AccountIdentifier sourceAccount) { this.sourceAccount = sourceAccount; }
    public void setDestinationAccount(AccountIdentifier destinationAccount) { this.destinationAccount = destinationAccount; }
    public void setExternalReferenceId(String externalReferenceId) { this.externalReferenceId = externalReferenceId; }
    public void setInternalReferenceId(String internalReferenceId) { this.internalReferenceId = internalReferenceId; }
    public void setTimestamp(LocalDateTime timestamp) { this.timestamp = timestamp; }
    public void setSettlementDate(LocalDateTime settlementDate) { this.settlementDate = settlementDate; }
    public void setStatus(TransactionStatus status) { this.status = status; }
    public void setDescription(String description) { this.description = description; }
    public void setMetadata(String metadata) { this.metadata = metadata; }
    public void setChannel(TransactionChannel channel) { this.channel = channel; }
    public void setInitiatorUserId(Long initiatorUserId) { this.initiatorUserId = initiatorUserId; }
    public void setApprovalStatus(String approvalStatus) { this.approvalStatus = approvalStatus; }
    public void setParentTransactionId(Long parentTransactionId) { this.parentTransactionId = parentTransactionId; }
}

/**
 * Represents a scheduled transaction for future processing.
 */
class ScheduledTransaction {
    private Long id;
    private String scheduleId; // Unique ID for the schedule
    private TransactionType type;
    private BigDecimal amount;
    private String currency;
    private Long sourceAccountId;
    private Long destinationAccountId; // Null for deposits/payments
    private String beneficiaryDetails; // For payments
    private String description;
    private LocalDateTime scheduledExecutionTime;
    private ScheduleStatus status;
    private String recurrencePattern; // E.g., "DAILY", "WEEKLY:MONDAY", "MONTHLY:15", "ONCE"
    private LocalDateTime nextExecutionTime;
    private LocalDateTime lastExecutionTime;
    private String lastExecutionResult; // E.g., "SUCCESS", "FAILED:InsufficientFunds"
    private int retryCount;
    private Long createdByUserId;
    private LocalDateTime creationDate;

    public ScheduledTransaction() {}
    public ScheduledTransaction(String scheduleId, TransactionType type, BigDecimal amount, String currency,
                                Long sourceAccountId, Long destinationAccountId, String beneficiaryDetails,
                                String description, LocalDateTime scheduledExecutionTime, String recurrencePattern, Long createdByUserId) {
        this.scheduleId = scheduleId;
        this.type = type;
        this.amount = amount;
        this.currency = currency;
        this.sourceAccountId = sourceAccountId;
        this.destinationAccountId = destinationAccountId;
```

```
        this.beneficiaryDetails = beneficiaryDetails;
        this.description = description;
        this.scheduledExecutionTime = scheduledExecutionTime;
        this.status = ScheduleStatus.PENDING;
        this.recurrencePattern = recurrencePattern;
        this.nextExecutionTime = scheduledExecutionTime;
        this.retryCount = 0;
        this.createdByUserId = createdByUserId;
        this.creationDate = LocalDateTime.now();
    }

    // Getters
    public Long getId() { return id; }
    public String getScheduleId() { return scheduleId; }
    public TransactionType getType() { return type; }
    public BigDecimal getAmount() { return amount; }
    public String getCurrency() { return currency; }
    public Long getSourceAccountId() { return sourceAccountId; }
    public Long getDestinationAccountId() { return destinationAccountId; }
    public String getBeneficiaryDetails() { return beneficiaryDetails; }
    public String getDescription() { return description; }
    public LocalDateTime getScheduledExecutionTime() { return scheduledExecutionTime; }
    public ScheduleStatus getStatus() { return status; }
    public String getRecurrencePattern() { return recurrencePattern; }
    public LocalDateTime getNextExecutionTime() { return nextExecutionTime; }
    public LocalDateTime getLastExecutionTime() { return lastExecutionTime; }
    public String getLastExecutionResult() { return lastExecutionResult; }
    public int getRetryCount() { return retryCount; }
    public Long getCreatedByUserId() { return createdByUserId; }
    public LocalDateTime getCreationDate() { return creationDate; }

    // Setters
    public void setId(Long id) { this.id = id; }
    public void setScheduleId(String scheduleId) { this.scheduleId = scheduleId; }
    public void setType(TransactionType type) { this.type = type; }
    public void setAmount(BigDecimal amount) { this.amount = amount; }
    public void setCurrency(String currency) { this.currency = currency; }
    public void setSourceAccountId(Long sourceAccountId) { this.sourceAccountId = sourceAccountId; }
    public void setDestinationAccountId(Long destinationAccountId) { this.destinationAccountId = destinationAccountId; }
    public void setBeneficiaryDetails(String beneficiaryDetails) { this.beneficiaryDetails = beneficiaryDetails; }
    public void setDescription(String description) { this.description = description; }
    public void setScheduledExecutionTime(LocalDateTime scheduledExecutionTime) { this.scheduledExecutionTime = scheduledExecutionTime; }
    public void setStatus(ScheduleStatus status) { this.status = status; }
    public void setRecurrencePattern(String recurrencePattern) { this.recurrencePattern = recurrencePattern; }
    public void setNextExecutionTime(LocalDateTime nextExecutionTime) { this.nextExecutionTime = nextExecutionTime; }
    public void setLastExecutionTime(LocalDateTime lastExecutionTime) { this.lastExecutionTime = lastExecutionTime; }
    public void setLastExecutionResult(String lastExecutionResult) { this.lastExecutionResult = lastExecutionResult; }
    public void setRetryCount(int retryCount) { this.retryCount = retryCount; }
    public void setCreatedByUserId(Long createdByUserId) { this.createdByUserId = createdByUserId; }
    public void setCreationDate(LocalDateTime creationDate) { this.creationDate = creationDate; }
}

/**
 * Entity for storing currency exchange rates.
 */
class CurrencyExchangeRate {
    private Long id;
    private String baseCurrency;
    private String targetCurrency;
    private BigDecimal rate;
    private LocalDateTime lastUpdateTime;

    public CurrencyExchangeRate() {}
    public CurrencyExchangeRate(Long id, String baseCurrency, String targetCurrency, BigDecimal rate) {
        this.id = id;
        this.baseCurrency = baseCurrency;
        this.targetCurrency = targetCurrency;
        this.rate = rate.setScale(4, GlobalConfig.CURRENCY_ROUNDING_MODE); // Rates need more precision
        this.lastUpdateTime = LocalDateTime.now();
    }

    // Getters
    public Long getId() { return id; }
    public String getBaseCurrency() { return baseCurrency; }
    public String getTargetCurrency() { return targetCurrency; }
    public BigDecimal getRate() { return rate; }
    public LocalDateTime getLastUpdateTime() { return lastUpdateTime; }

    // Setters
    public void setId(Long id) { this.id = id; }
    public void setBaseCurrency(String baseCurrency) { this.baseCurrency = baseCurrency; }
    public void setTargetCurrency(String targetCurrency) { this.targetCurrency = targetCurrency; }
    public void setRate(BigDecimal rate) {
        this.rate = rate.setScale(4, GlobalConfig.CURRENCY_ROUNDING_MODE);
        this.lastUpdateTime = LocalDateTime.now();
    }
    public void setLastUpdateTime(LocalDateTime lastUpdateTime) { this.lastUpdateTime = lastUpdateTime; }
}

/**
 * Entity for storing fee configurations.
 */
class FeeConfiguration {
    private Long id;
    private TransactionType transactionType;
    private String currency; // Applies to specific currency, or null for all
    private BigDecimal minAmount;
    private BigDecimal maxAmount;
    private BigDecimal fixedFee;
    private BigDecimal percentageFee; // e.g., 0.01 for 1%
    private String description;
    private boolean isActive;

    public FeeConfiguration() {}
    public FeeConfiguration(Long id, TransactionType transactionType, String currency, BigDecimal minAmount, BigDecimal maxAmount, BigDecimal fixedFee, BigDecimal percentageFee, String description, boolean isActive) {
        this.id = id;
        this.transactionType = transactionType;
        this.currency = currency;
        this.minAmount = minAmount;
        this.maxAmount = maxAmount;
        this.fixedFee = fixedFee.setScale(GlobalConfig.CURRENCY_SCALE, GlobalConfig.CURRENCY_ROUNDING_MODE);
        this.percentageFee = percentageFee;
        this.description = description;
        this.isActive = isActive;
    }

    // Getters
    public Long getId() { return id; }
    public TransactionType getTransactionType() { return transactionType; }
    public String getCurrency() { return currency; }
    public BigDecimal getMinAmount() { return minAmount; }
    public BigDecimal getMaxAmount() { return maxAmount; }
    public BigDecimal getFixedFee() { return fixedFee; }
    public BigDecimal getPercentageFee() { return percentageFee; }
    public String getDescription() { return description; }
    public boolean isActive() { return isActive; }

    // Setters
    public void setId(Long id) { this.id = id; }
```

```
    public void setTransactionType(TransactionType transactionType) { this.transactionType = transactionType; }
    public void setCurrency(String currency) { this.currency = currency; }
    public void setMinAmount(BigDecimal minAmount) { this.minAmount = minAmount; }
    public void setMaxAmount(BigDecimal maxAmount) { this.maxAmount = maxAmount; }
    public void setFixedFee(BigDecimal fixedFee) { this.fixedFee = fixedFee.setScale(GlobalConfig.CURRENCY_SCALE, GlobalConfig.CURRENCY_ROUNDING_MODE); }
    public void setPercentageFee(BigDecimal percentageFee) { this.percentageFee = percentageFee; }
    public void setDescription(String description) { this.description = description; }
    public void setActive(boolean active) { isActive = active; }
}


/**
 * Data Transfer Object (DTO) for Deposit requests.
 */
class DepositRequest {
    private String transactionId;
    private Long accountId;
    private BigDecimal amount;
    private String currency;
    private String description;
    private TransactionChannel channel;
    private Long initiatorUserId;
    private String metadata;

    public DepositRequest() {}
    public DepositRequest(String transactionId, Long accountId, BigDecimal amount, String currency, String description, TransactionChannel channel, Long initiatorUserId, String metadata) {
        this.transactionId = transactionId;
        this.accountId = accountId;
        this.amount = amount;
        this.currency = currency;
        this.description = description;
        this.channel = channel;
        this.initiatorUserId = initiatorUserId;
        this.metadata = metadata;
    }

    // Getters
    public String getTransactionId() { return transactionId; }
    public Long getAccountId() { return accountId; }
    public BigDecimal getAmount() { return amount; }
    public String getCurrency() { return currency; }
    public String getDescription() { return description; }
    public TransactionChannel getChannel() { return channel; }
    public Long getInitiatorUserId() { return initiatorUserId; }
    public String getMetadata() { return metadata; }

    // Setters (for potential deserialization by REST frameworks)
    public void setTransactionId(String transactionId) { this.transactionId = transactionId; }
    public void setAccountId(Long accountId) { this.accountId = accountId; }
    public void setAmount(BigDecimal amount) { this.amount = amount; }
    public void setCurrency(String currency) { this.currency = currency; }
    public void setDescription(String description) { this.description = description; }
    public void setChannel(TransactionChannel channel) { this.channel = channel; }
    public void setInitiatorUserId(Long initiatorUserId) { this.initiatorUserId = initiatorUserId; }
    public void setMetadata(String metadata) { this.metadata = metadata; }
}

/**
 * Data Transfer Object (DTO) for Payment requests.
 */
class PaymentRequest {
    private String transactionId;
    private Long sourceAccountId;
    private BigDecimal amount;
    private String currency;
    private String beneficiaryDetails; // Details about the external party receiving the payment
    private String description;
    private TransactionChannel channel;
    private Long initiatorUserId;
    private String metadata;

    public PaymentRequest() {}
    public PaymentRequest(String transactionId, Long sourceAccountId, BigDecimal amount, String currency, String beneficiaryDetails, String description, TransactionChannel channel, Long initiatorUserId, String metadata) {
        this.transactionId = transactionId;
        this.sourceAccountId = sourceAccountId;
        this.amount = amount;
        this.currency = currency;
        this.beneficiaryDetails = beneficiaryDetails;
        this.description = description;
        this.channel = channel;
        this.initiatorUserId = initiatorUserId;
        this.metadata = metadata;
    }

    // Getters
    public String getTransactionId() { return transactionId; }
    public Long getSourceAccountId() { return sourceAccountId; }
    public BigDecimal getAmount() { return amount; }
    public String getCurrency() { return currency; }
    public String getBeneficiaryDetails() { return beneficiaryDetails; }
    public String getDescription() { return description; }
    public TransactionChannel getChannel() { return channel; }
    public Long getInitiatorUserId() { return initiatorUserId; }
    public String getMetadata() { return metadata; }

    // Setters
    public void setTransactionId(String transactionId) { this.transactionId = transactionId; }
    public void setSourceAccountId(Long sourceAccountId) { this.sourceAccountId = sourceAccountId; }
    public void setAmount(BigDecimal amount) { this.amount = amount; }
    public void setCurrency(String currency) { this.currency = currency; }
    public void setBeneficiaryDetails(String beneficiaryDetails) { this.beneficiaryDetails = beneficiaryDetails; }
    public void setDescription(String description) { this.description = description; }
    public void setChannel(TransactionChannel channel) { this.channel = channel; }
    public void setInitiatorUserId(Long initiatorUserId) { this.initiatorUserId = initiatorUserId; }
    public void setMetadata(String metadata) { this.metadata = metadata; }
}

/**
 * Data Transfer Object (DTO) for Transfer requests.
 */
class TransferRequest {
    private String transactionId;
    private Long sourceAccountId;
    private Long destinationAccountId;
    private BigDecimal amount;
    private String currency;
    private String description;
    private TransactionChannel channel;
    private Long initiatorUserId;
    private String metadata;

    public TransferRequest() {}
    public TransferRequest(String transactionId, Long sourceAccountId, Long destinationAccountId, BigDecimal amount, String currency, String description, TransactionChannel channel, Long initiatorUserId, String metadata) {
        this.transactionId = transactionId;
        this.sourceAccountId = sourceAccountId;
        this.destinationAccountId = destinationAccountId;
        this.amount = amount;
        this.currency = currency;
        this.description = description;
```

```
        this.channel = channel;
        this.initiatorUserId = initiatorUserId;
        this.metadata = metadata;
    }

    // Getters
    public String getTransactionId() { return transactionId; }
    public Long getSourceAccountId() { return sourceAccountId; }
    public Long getDestinationAccountId() { return destinationAccountId; }
    public BigDecimal getAmount() { return amount; }
    public String getCurrency() { return currency; }
    public String getDescription() { return description; }
    public TransactionChannel getChannel() { return channel; }
    public Long getInitiatorUserId() { return initiatorUserId; }
    public String getMetadata() { return metadata; }

    // Setters
    public void setTransactionId(String transactionId) { this.transactionId = transactionId; }
    public void setSourceAccountId(Long sourceAccountId) { this.sourceAccountId = sourceAccountId; }
    public void setDestinationAccountId(Long destinationAccountId) { this.destinationAccountId = destinationAccountId; }
    public void setAmount(BigDecimal amount) { this.amount = amount; }
    public void setCurrency(String currency) { this.currency = currency; }
    public void setDescription(String description) { this.description = description; }
    public void setChannel(TransactionChannel channel) { this.channel = channel; }
    public void setInitiatorUserId(Long initiatorUserId) { this.initiatorUserId = initiatorUserId; }
    public void setMetadata(String metadata) { this.metadata = metadata; }
}

/**
 * Data Transfer Object (DTO) for Refund requests.
 */
class RefundRequest {
    private String transactionId; // Unique ID for the refund request itself
    private String originalTransactionId; // The ID of the transaction to be refunded
    private BigDecimal amount; // Optional, if null, refund full amount
    private String description;
    private TransactionChannel channel;
    private Long initiatorUserId;

    public RefundRequest() {}
    public RefundRequest(String transactionId, String originalTransactionId, BigDecimal amount, String description, TransactionChannel channel, Long initiatorUserId) {
        this.transactionId = transactionId;
        this.originalTransactionId = originalTransactionId;
        this.amount = amount;
        this.description = description;
        this.channel = channel;
        this.initiatorUserId = initiatorUserId;
    }

    public String getTransactionId() { return transactionId; }
    public String getOriginalTransactionId() { return originalTransactionId; }
    public BigDecimal getAmount() { return amount; }
    public String getDescription() { return description; }
    public TransactionChannel getChannel() { return channel; }
    public Long getInitiatorUserId() { return initiatorUserId; }

    public void setTransactionId(String transactionId) { this.transactionId = transactionId; }
    public void setOriginalTransactionId(String originalTransactionId) { this.originalTransactionId = originalTransactionId; }
    public void setAmount(BigDecimal amount) { this.amount = amount; }
    public void setDescription(String description) { this.description = description; }
    public void setChannel(TransactionChannel channel) { this.channel = channel; }
    public void setInitiatorUserId(Long initiatorUserId) { this.initiatorUserId = initiatorUserId; }
}

/**
 * Data Transfer Object (DTO) for Reversal requests.
 */
class ReversalRequest {
    private String transactionId; // Unique ID for the reversal request itself
    private String originalTransactionId; // The ID of the transaction to be reversed
    private String reason;
    private TransactionChannel channel;
    private Long initiatorUserId;

    public ReversalRequest() {}
    public ReversalRequest(String transactionId, String originalTransactionId, String reason, TransactionChannel channel, Long initiatorUserId) {
        this.transactionId = transactionId;
        this.originalTransactionId = originalTransactionId;
        this.reason = reason;
        this.channel = channel;
        this.initiatorUserId = initiatorUserId;
    }

    public String getTransactionId() { return transactionId; }
    public String getOriginalTransactionId() { return originalTransactionId; }
    public String getReason() { return reason; }
    public TransactionChannel getChannel() { return channel; }
    public Long getInitiatorUserId() { return initiatorUserId; }

    public void setTransactionId(String transactionId) { this.transactionId = transactionId; }
    public void setOriginalTransactionId(String originalTransactionId) { this.originalTransactionId = originalTransactionId; }
    public void setReason(String reason) { this.reason = reason; }
    public void setChannel(TransactionChannel channel) { this.channel = channel; }
    public void setInitiatorUserId(Long initiatorUserId) { this.initiatorUserId = initiatorUserId; }
}

/**
 * DTO for setting up a scheduled transaction.
 */
class ScheduledTransactionRequest {
    private String scheduleId; // Client-provided or generated
    private TransactionType type; // DEPOSIT, PAYMENT, TRANSFER
    private BigDecimal amount;
    private String currency;
    private Long sourceAccountId;
    private Long destinationAccountId; // Optional
    private String beneficiaryDetails; // Optional for Payments
    private String description;
    private LocalDateTime scheduledExecutionTime;
    private String recurrencePattern; // e.g., "ONCE", "DAILY", "WEEKLY:MONDAY", "MONTHLY:15"
    private Long createdByUserId;

    public ScheduledTransactionRequest() {}
    public ScheduledTransactionRequest(String scheduleId, TransactionType type, BigDecimal amount, String currency, Long sourceAccountId, Long destinationAccountId, String beneficiaryDetails, String description, LocalDateTime scheduledExecutionTime, String recurrencePattern, Long createdByUserId) {
        this.scheduleId = scheduleId;
        this.type = type;
        this.amount = amount;
        this.currency = currency;
        this.sourceAccountId = sourceAccountId;
        this.destinationAccountId = destinationAccountId;
        this.beneficiaryDetails = beneficiaryDetails;
        this.description = description;
        this.scheduledExecutionTime = scheduledExecutionTime;
        this.recurrencePattern = recurrencePattern;
        this.createdByUserId = createdByUserId;
    }

    public String getScheduleId() { return scheduleId; }
    public TransactionType getType() { return type; }
```

```
    public BigDecimal getAmount() { return amount; }
    public String getCurrency() { return currency; }
    public Long getSourceAccountId() { return sourceAccountId; }
    public Long getDestinationAccountId() { return destinationAccountId; }
    public String getBeneficiaryDetails() { return beneficiaryDetails; }
    public String getDescription() { return description; }
    public LocalDateTime getScheduledExecutionTime() { return scheduledExecutionTime; }
    public String getRecurrencePattern() { return recurrencePattern; }
    public Long getCreatedByUserId() { return createdByUserId; }

    public void setScheduleId(String scheduleId) { this.scheduleId = scheduleId; }
    public void setType(TransactionType type) { this.type = type; }
    public void setAmount(BigDecimal amount) { this.amount = amount; }
    public void setCurrency(String currency) { this.currency = currency; }
    public void setSourceAccountId(Long sourceAccountId) { this.sourceAccountId = sourceAccountId; }
    public void setDestinationAccountId(Long destinationAccountId) { this.destinationAccountId = destinationAccountId; }
    public void setBeneficiaryDetails(String beneficiaryDetails) { this.beneficiaryDetails = beneficiaryDetails; }
    public void setDescription(String description) { this.description = description; }
    public void setScheduledExecutionTime(LocalDateTime scheduledExecutionTime) { this.scheduledExecutionTime = scheduledExecutionTime; }
    public void setRecurrencePattern(String recurrencePattern) { this.recurrencePattern = recurrencePattern; }
    public void setCreatedByUserId(Long createdByUserId) { this.createdByUserId = createdByUserId; }
}

/**
 * Data Transfer Object (DTO) for Transaction responses.
 */
class TransactionResponse {
    private String transactionId; // Client-provided or generated
    private String systemTransactionId; // Internally unique ID
    private TransactionStatus status;
    private String message;
    private BigDecimal newBalance; // For the primary account involved
    private String newBalanceCurrency;
    private BigDecimal appliedFees;
    private String processedCurrency;
    private BigDecimal processedAmount;
    private String approvalStatus; // If applicable

    public TransactionResponse() {}
    public TransactionResponse(String transactionId, String systemTransactionId, TransactionStatus status, String message, BigDecimal newBalance, String newBalanceCurrency, BigDecimal appliedFees, BigDecimal processedAmount, String processedCurrency, String approvalStatus) {
        this.transactionId = transactionId;
        this.systemTransactionId = systemTransactionId;
        this.status = status;
        this.message = message;
        this.newBalance = newBalance != null ? newBalance.setScale(GlobalConfig.CURRENCY_SCALE, GlobalConfig.CURRENCY_ROUNDING_MODE) : null;
        this.newBalanceCurrency = newBalanceCurrency;
        this.appliedFees = appliedFees != null ? appliedFees.setScale(GlobalConfig.CURRENCY_SCALE, GlobalConfig.CURRENCY_ROUNDING_MODE) : null;
        this.processedAmount = processedAmount != null ? processedAmount.setScale(GlobalConfig.CURRENCY_SCALE, GlobalConfig.CURRENCY_ROUNDING_MODE) : null;
        this.processedCurrency = processedCurrency;
        this.approvalStatus = approvalStatus;
    }

    // Getters
    public String getTransactionId() { return transactionId; }
    public String getSystemTransactionId() { return systemTransactionId; }
    public TransactionStatus getStatus() { return status; }
    public String getMessage() { return message; }
    public BigDecimal getNewBalance() { return newBalance; }
    public String getNewBalanceCurrency() { return newBalanceCurrency; }
    public BigDecimal getAppliedFees() { return appliedFees; }
    public BigDecimal getProcessedAmount() { return processedAmount; }
    public String getProcessedCurrency() { return processedCurrency; }
    public String getApprovalStatus() { return approvalStatus; }

    // Setters
    public void setTransactionId(String transactionId) { this.transactionId = transactionId; }
    public void setSystemTransactionId(String systemTransactionId) { this.systemTransactionId = systemTransactionId; }
    public void setStatus(TransactionStatus status) { this.status = status; }
    public void setMessage(String message) { this.message = message; }
    public void setNewBalance(BigDecimal newBalance) { this.newBalance = newBalance != null ? newBalance.setScale(GlobalConfig.CURRENCY_SCALE, GlobalConfig.CURRENCY_ROUNDING_MODE) : null; }
    public void setNewBalanceCurrency(String newBalanceCurrency) { this.newBalanceCurrency = newBalanceCurrency; }
    public void setAppliedFees(BigDecimal appliedFees) { this.appliedFees = appliedFees != null ? appliedFees.setScale(GlobalConfig.CURRENCY_SCALE, GlobalConfig.CURRENCY_ROUNDING_MODE) : null; }
    public void setProcessedAmount(BigDecimal processedAmount) { this.processedAmount = processedAmount != null ? processedAmount.setScale(GlobalConfig.CURRENCY_SCALE, GlobalConfig.CURRENCY_ROUNDING_MODE) : null; }
    public void setProcessedCurrency(String processedCurrency) { this.processedCurrency = processedCurrency; }
    public void setApprovalStatus(String approvalStatus) { this.approvalStatus = approvalStatus; }
}

/**
 * DTO for querying transactions.
 */
class TransactionQueryRequest {
    private Long accountId;
    private TransactionType type;
    private TransactionStatus status;
    private LocalDateTime startDate;
    private LocalDateTime endDate;
    private String currency;
    private int page = 0;
    private int size = 20;

    public TransactionQueryRequest() {}
    public TransactionQueryRequest(Long accountId, TransactionType type, TransactionStatus status, LocalDateTime startDate, LocalDateTime endDate, String currency, int page, int size) {
        this.accountId = accountId;
        this.type = type;
        this.status = status;
        this.startDate = startDate;
        this.endDate = endDate;
        this.currency = currency;
        this.page = page;
        this.size = size;
    }

    public Long getAccountId() { return accountId; }
    public TransactionType getType() { return type; }
    public TransactionStatus getStatus() { return status; }
    public LocalDateTime getStartDate() { return startDate; }
    public LocalDateTime getEndDate() { return endDate; }
    public String getCurrency() { return currency; }
    public int getPage() { return page; }
    public int getSize() { return size; }

    public void setAccountId(Long accountId) { this.accountId = accountId; }
    public void setType(TransactionType type) { this.type = type; }
    public void setStatus(TransactionStatus status) { this.status = status; }
    public void setStartDate(LocalDateTime startDate) { this.startDate = startDate; }
    public void setEndDate(LocalDateTime endDate) { this.endDate = endDate; }
    public void setCurrency(String currency) { this.currency = currency; }
    public void setPage(int page) { this.page = page; }
    public void setSize(int size) { this.size = size; }
}

/**
 * DTO for transaction report generation requests.
 */
class TransactionReportRequest {
    private Long accountId;
    private LocalDateTime startDate;
    private LocalDateTime endDate;
    private List<TransactionType> types;
```

```
    private List<TransactionStatus> statuses;
    private String format; // e.g., "CSV", "PDF" (mocked)

    public TransactionReportRequest() {}
    public TransactionReportRequest(Long accountId, LocalDateTime startDate, LocalDateTime endDate, List<TransactionType> types, List<TransactionStatus> statuses, String format) {
        this.accountId = accountId;
        this.startDate = startDate;
        this.endDate = endDate;
        this.types = types;
        this.statuses = statuses;
        this.format = format;
    }

    // Getters and Setters
    public Long getAccountId() { return accountId; }
    public void setAccountId(Long accountId) { this.accountId = accountId; }
    public LocalDateTime getStartDate() { return startDate; }
    public void setStartDate(LocalDateTime startDate) { this.startDate = startDate; }
    public LocalDateTime getEndDate() { return endDate; }
    public void setEndDate(LocalDateTime endDate) { this.endDate = endDate; }
    public List<TransactionType> getTypes() { return types; }
    public void setTypes(List<TransactionType> types) { this.types = types; }
    public List<TransactionStatus> getStatuses() { return statuses; }
    public void setStatuses(List<TransactionStatus> statuses) { this.statuses = statuses; }
    public String getFormat() { return format; }
    public void setFormat(String format) { this.format = format; }
}

/**
 * DTO for transaction report responses.
 */
class TransactionReportResponse {
    private String reportId;
    private String status; // e.g., "GENERATED", "PENDING", "FAILED"
    private String downloadLink; // Mocked
    private LocalDateTime generationTime;

    public TransactionReportResponse() {}
    public TransactionReportResponse(String reportId, String status, String downloadLink, LocalDateTime generationTime) {
        this.reportId = reportId;
        this.status = status;
        this.downloadLink = downloadLink;
        this.generationTime = generationTime;
    }

    public String getReportId() { return reportId; }
    public void setReportId(String reportId) { this.reportId = reportId; }
    public String getStatus() { return status; }
    public void setStatus(String status) { this.status = status; }
    public String getDownloadLink() { return downloadLink; }
    public void setDownloadLink(String downloadLink) { this.downloadLink = downloadLink; }
    public LocalDateTime getGenerationTime() { return generationTime; }
    public void setGenerationTime(LocalDateTime generationTime) { this.generationTime = generationTime; }
}

// endregion

// region Custom Exception classes for specific error conditions (Expanded)
class AccountNotFoundException extends RuntimeException {
    public AccountNotFoundException(String message) { super(message); }
    public AccountNotFoundException(String message, Throwable cause) { super(message, cause); }
}
class InsufficientFundsException extends RuntimeException {
    public InsufficientFundsException(String message) { super(message); }
}
class InvalidTransactionException extends RuntimeException {
    public InvalidTransactionException(String message) { super(message); }
}
class TransactionAlreadyProcessedException extends RuntimeException {
    public TransactionAlreadyProcessedException(String message) { super(message); }
}
class ConcurrentAccountModificationException extends RuntimeException {
    public ConcurrentAccountModificationException(String message) { super(message); }
}
class FraudDetectionException extends RuntimeException {
    public FraudDetectionException(String message) { super(message); }
}
class CurrencyMismatchException extends RuntimeException {
    public CurrencyMismatchException(String message) { super(message); }
}
class ExchangeRateNotFoundException extends RuntimeException {
    public ExchangeRateNotFoundException(String message) { super(message); }
}
class FeeConfigurationException extends RuntimeException {
    public FeeConfigurationException(String message) { super(message); }
}
class ScheduledTransactionException extends RuntimeException {
    public ScheduledTransactionException(String message) { super(message); }
}
class RateLimitExceededException extends RuntimeException {
    public RateLimitExceededException(String message) { super(message); }
}
class AuthorizationRequiredException extends RuntimeException {
    public AuthorizationRequiredException(String message) { super(message); }
}
class BadGatewayException extends RuntimeException { // For external system failures
    public BadGatewayException(String message) { super(message); }
    public BadGatewayException(String message, Throwable cause) { super(message, cause); }
}
// endregion

// region Repository Interfaces (Expanded)
interface AccountRepository {
    Optional<Account> findById(Long id);
    Optional<Account> findByAccountNumber(String accountNumber);
    Account save(Account account);
    Account saveWithOptimisticLocking(Account account, Long expectedVersion);
    List<Account> findByAccountHolderId(Long accountHolderId);
    List<Account> findAll();
}

interface TransactionRepository {
    Optional<Transaction> findByTransactionId(String transactionId); // Idempotency key
    Optional<Transaction> findBySystemTransactionId(String systemTransactionId); // Internal unique ID
    Transaction save(Transaction transaction);
    List<Transaction> findBySourceAccountId(Long sourceAccountId);
    List<Transaction> findByDestinationAccountId(Long destinationAccountId);
    List<Transaction> findByAccountIdAndDateRange(Long accountId, LocalDateTime start, LocalDateTime end);
    List<Transaction> findFilteredTransactions(TransactionQueryRequest query);
    List<Transaction> findByParentTransactionId(Long parentTransactionId);
}

interface AccountHolderRepository {
    Optional<AccountHolder> findById(Long id);
    Optional<AccountHolder> findByUserId(String userId);
    AccountHolder save(AccountHolder accountHolder);
}

interface CurrencyExchangeRateRepository {
```

```
    Optional<CurrencyExchangeRate> findByBaseCurrencyAndTargetCurrency(String baseCurrency, String targetCurrency);
    CurrencyExchangeRate save(CurrencyExchangeRate rate);
    List<CurrencyExchangeRate> findAll();
}

interface FeeConfigurationRepository {
    List<FeeConfiguration> findByTransactionTypeAndCurrency(TransactionType type, String currency);
    List<FeeConfiguration> findByTransactionType(TransactionType type);
    FeeConfiguration save(FeeConfiguration config);
    Optional<FeeConfiguration> findById(Long id);
    List<FeeConfiguration> findAll();
}

interface ScheduledTransactionRepository {
    Optional<ScheduledTransaction> findByScheduleId(String scheduleId);
    ScheduledTransaction save(ScheduledTransaction scheduledTransaction);
    List<ScheduledTransaction> findByStatusAndNextExecutionTimeBefore(ScheduleStatus status, LocalDateTime dateTime);
    List<ScheduledTransaction> findAll();
}
// endregion

// region In-memory Repository Implementations (Expanded & Simplified)
@Component
class InMemoryAccountRepository implements AccountRepository {
    private static final Logger logger = LoggerFactory.getLogger(InMemoryAccountRepository.class);
    private final Map<Long, Account> accounts = new ConcurrentHashMap<>();
    private final AtomicLong idGenerator = new AtomicLong();

    public InMemoryAccountRepository() {
        // Initialize with some dummy data for testing
        Account acc1 = new Account(idGenerator.incrementAndGet(), "ACC001", 101L, AccountType.CHECKING, new BigDecimal("10000.00"), "USD", AccountStatus.ACTIVE, GlobalConfig.MIN_ACCOUNT_BALANCE);
        Account acc2 = new Account(idGenerator.incrementAndGet(), "ACC002", 101L, AccountType.SAVINGS, new BigDecimal("5000.00"), "USD", AccountStatus.ACTIVE, new BigDecimal("100.00"));
        Account acc3 = new Account(idGenerator.incrementAndGet(), "ACC003", 102L, AccountType.CHECKING, new BigDecimal("2000.00"), "EUR", AccountStatus.ACTIVE, GlobalConfig.MIN_ACCOUNT_BALANCE);
        accounts.put(acc1.getId(), acc1);
        accounts.put(acc2.getId(), acc2);
        accounts.put(acc3.getId(), acc3);
        logger.info("Initialized InMemoryAccountRepository with {} accounts.", accounts.size());
    }

    @Override
    public Optional<Account> findById(Long id) {
        return Optional.ofNullable(accounts.get(id));
    }

    @Override
    public Optional<Account> findByAccountNumber(String accountNumber) {
        return accounts.values().stream()
                .filter(acc -> acc.getAccountNumber().equals(accountNumber))
                .findFirst();
    }

    @Override
    public Account save(Account account) {
        if (account.getId() == null) {
            account.setId(idGenerator.incrementAndGet());
            account.setCreationDate(LocalDateTime.now());
            account.setVersion(0L);
        } else {
            // Simulate update timestamp without optimistic locking for this method
            account.setLastUpdateDate(LocalDateTime.now());
            account.incrementVersion(); // Increment version on every save
        }
        accounts.put(account.getId(), account);
        logger.debug("Saved account: {}", account.getId());
        return account;
    }

    @Override
    public Account saveWithOptimisticLocking(Account account, Long expectedVersion) {
        Account currentAccount = accounts.get(account.getId());
        if (currentAccount == null) {
            if (account.getId() == null) { // New account
                account.setId(idGenerator.incrementAndGet());
                account.setCreationDate(LocalDateTime.now());
                account.setLastUpdateDate(LocalDateTime.now());
                account.setVersion(0L);
            } else { // Trying to update non-existent account
                throw new AccountNotFoundException("Account not found with ID: " + account.getId());
            }
        } else {
            if (!currentAccount.getVersion().equals(expectedVersion)) {
                logger.warn("Optimistic locking conflict for account ID {}. Expected version: {}, Actual version: {}", account.getId(), expectedVersion, currentAccount.getVersion());
                throw new ConcurrentAccountModificationException("Account ID " + account.getId() + " was modified concurrently.");
            }
            // Update fields from the provided account object, then increment version
            currentAccount.setAccountNumber(account.getAccountNumber());
            currentAccount.setAccountHolderId(account.getAccountHolderId());
            currentAccount.setType(account.getType());
            currentAccount.setBalance(account.getBalance());
            currentAccount.setCurrency(account.getCurrency());
            currentAccount.setStatus(account.getStatus());
            currentAccount.setMinimumBalance(account.getMinimumBalance());
            // currentAccount.setCreationDate(account.getCreationDate()); // Should not change
            currentAccount.setLastUpdateDate(LocalDateTime.now());
            currentAccount.incrementVersion();
            account = currentAccount; // Ensure the caller gets the updated object with incremented version
        }
        accounts.put(account.getId(), account);
        logger.debug("Saved account with optimistic locking: {}", account.getId());
        return account;
    }

    @Override
    public List<Account> findByAccountHolderId(Long accountHolderId) {
        return accounts.values().stream()
                .filter(acc -> acc.getAccountHolderId() != null && acc.getAccountHolderId().equals(accountHolderId))
                .collect(Collectors.toList());
    }

    @Override
    public List<Account> findAll() {
        return List.copyOf(accounts.values());
    }
}

@Component
class InMemoryTransactionRepository implements TransactionRepository {
    private final Map<Long, Transaction> transactions = new ConcurrentHashMap<>();
    private final Map<String, Transaction> transactionsByIdempotencyKey = new ConcurrentHashMap<>(); // For client-provided ID
    private final Map<String, Transaction> transactionsBySystemId = new ConcurrentHashMap<>(); // For internally generated UUID
    private final AtomicLong idGenerator = new AtomicLong();

    @Override
    public Optional<Transaction> findByTransactionId(String transactionId) {
        return Optional.ofNullable(transactionsByIdempotencyKey.get(transactionId));
    }

    @Override
    public Optional<Transaction> findBySystemTransactionId(String systemTransactionId) {
```

```
        return Optional.ofNullable(transactionsBySystemId.get(systemTransactionId));
    }

    @Override
    public Transaction save(Transaction transaction) {
        if (transaction.getId() == null) {
            transaction.setId(idGenerator.incrementAndGet());
            if (transaction.getSystemTransactionId() == null || transaction.getSystemTransactionId().isEmpty()) {
                transaction.setSystemTransactionId(UUID.randomUUID().toString()); // Ensure system ID
            }
        }
        transactions.put(transaction.getId(), transaction);
        if (transaction.getTransactionId() != null && !transaction.getTransactionId().isEmpty()) {
            transactionsByIdempotencyKey.put(transaction.getTransactionId(), transaction);
        }
        transactionsBySystemId.put(transaction.getSystemTransactionId(), transaction);
        return transaction;
    }

    @Override
    public List<Transaction> findBySourceAccountId(Long sourceAccountId) {
        return transactions.values().stream()
                .filter(tx -> tx.getSourceAccount() != null && Objects.equals(tx.getSourceAccount().getAccountId(), sourceAccountId))
                .collect(Collectors.toList());
    }

    @Override
    public List<Transaction> findByDestinationAccountId(Long destinationAccountId) {
        return transactions.values().stream()
                .filter(tx -> tx.getDestinationAccount() != null && Objects.equals(tx.getDestinationAccount().getAccountId(), destinationAccountId))
                .collect(Collectors.toList());
    }

    @Override
    public List<Transaction> findByAccountIdAndDateRange(Long accountId, LocalDateTime start, LocalDateTime end) {
        return transactions.values().stream()
                .filter(tx -> (tx.getSourceAccount() != null && Objects.equals(tx.getSourceAccount().getAccountId(), accountId)) ||
                              (tx.getDestinationAccount() != null && Objects.equals(tx.getDestinationAccount().getAccountId(), accountId)))
                .filter(tx -> !tx.getTimestamp().isBefore(start) && !tx.getTimestamp().isAfter(end))
                .sorted(Comparator.comparing(Transaction::getTimestamp).reversed())
                .collect(Collectors.toList());
    }

    @Override
    public List<Transaction> findFilteredTransactions(TransactionQueryRequest query) {
        Stream<Transaction> filteredStream = transactions.values().stream();

        if (query.getAccountId() != null) {
            filteredStream = filteredStream.filter(tx ->
                    (tx.getSourceAccount() != null && Objects.equals(tx.getSourceAccount().getAccountId(), query.getAccountId())) ||
                    (tx.getDestinationAccount() != null && Objects.equals(tx.getDestinationAccount().getAccountId(), query.getAccountId())));
        }
        if (query.getType() != null) {
            filteredStream = filteredStream.filter(tx -> tx.getType() == query.getType());
        }
        if (query.getStatus() != null) {
            filteredStream = filteredStream.filter(tx -> tx.getStatus() == query.getStatus());
        }
        if (query.getStartDate() != null) {
            filteredStream = filteredStream.filter(tx -> !tx.getTimestamp().isBefore(query.getStartDate()));
        }
        if (query.getEndDate() != null) {
            filteredStream = filteredStream.filter(tx -> !tx.getTimestamp().isAfter(query.getEndDate()));
        }
        if (query.getCurrency() != null && !query.getCurrency().isEmpty()) {
            filteredStream = filteredStream.filter(tx ->
                    (tx.getOriginalCurrency() != null && tx.getOriginalCurrency().equalsIgnoreCase(query.getCurrency())) ||
                    (tx.getProcessedCurrency() != null && tx.getProcessedCurrency().equalsIgnoreCase(query.getCurrency())));
        }

        return filteredStream
                .sorted(Comparator.comparing(Transaction::getTimestamp).reversed())
                .skip((long) query.getPage() * query.getSize())
                .limit(query.getSize())
                .collect(Collectors.toList());
    }

    @Override
    public List<Transaction> findByParentTransactionId(Long parentTransactionId) {
        return transactions.values().stream()
                .filter(tx -> Objects.equals(tx.getParentTransactionId(), parentTransactionId))
                .collect(Collectors.toList());
    }
}

@Component
class InMemoryAccountHolderRepository implements AccountHolderRepository {
    private final Map<Long, AccountHolder> accountHolders = new ConcurrentHashMap<>();
    private final AtomicLong idGenerator = new AtomicLong();

    public InMemoryAccountHolderRepository() {
        AccountHolder ah1 = new AccountHolder(idGenerator.incrementAndGet(), "user123", "Alice", "Smith", "alice@example.com", "111-222-3333", "VERIFIED");
        AccountHolder ah2 = new AccountHolder(idGenerator.incrementAndGet(), "user456", "Bob", "Johnson", "bob@example.com", "444-555-6666", "PENDING");
        accountHolders.put(ah1.getId(), ah1);
        accountHolders.put(ah2.getId(), ah2);
    }

    @Override
    public Optional<AccountHolder> findById(Long id) {
        return Optional.ofNullable(accountHolders.get(id));
    }

    @Override
    public Optional<AccountHolder> findByUserId(String userId) {
        return accountHolders.values().stream()
                .filter(ah -> ah.getUserId().equals(userId))
                .findFirst();
    }

    @Override
    public AccountHolder save(AccountHolder accountHolder) {
        if (accountHolder.getId() == null) {
            accountHolder.setId(idGenerator.incrementAndGet());
            accountHolder.setRegistrationDate(LocalDateTime.now());
        }
        accountHolders.put(accountHolder.getId(), accountHolder);
        return accountHolder;
    }
}

@Component
class InMemoryCurrencyExchangeRateRepository implements CurrencyExchangeRateRepository {
    private final Map<String, CurrencyExchangeRate> rates = new ConcurrentHashMap<>(); // Key: BASE_TARGET
    private final AtomicLong idGenerator = new AtomicLong();

    public InMemoryCurrencyExchangeRateRepository() {
        rates.put("USD_EUR", new CurrencyExchangeRate(idGenerator.incrementAndGet(), "USD", "EUR", new BigDecimal("0.85")));
        rates.put("EUR_USD", new CurrencyExchangeRate(idGenerator.incrementAndGet(), "EUR", "USD", new BigDecimal("1.17")));
        rates.put("USD_GBP", new CurrencyExchangeRate(idGenerator.incrementAndGet(), "USD", "GBP", new BigDecimal("0.75")));
```

```
        rates.put("GBP_USD", new CurrencyExchangeRate(idGenerator.incrementAndGet(), "GBP", "USD", new BigDecimal("1.33")));
        // Add direct rates for simplicity, in reality it might go through a base like USD
        rates.put("EUR_GBP", new CurrencyExchangeRate(idGenerator.incrementAndGet(), "EUR", "GBP", new BigDecimal("0.88")));
        rates.put("GBP_EUR", new CurrencyExchangeRate(idGenerator.incrementAndGet(), "GBP", "EUR", new BigDecimal("1.13")));
    }

    @Override
    public Optional<CurrencyExchangeRate> findByBaseCurrencyAndTargetCurrency(String baseCurrency, String targetCurrency) {
        return Optional.ofNullable(rates.get(baseCurrency + "_" + targetCurrency));
    }

    @Override
    public CurrencyExchangeRate save(CurrencyExchangeRate rate) {
        if (rate.getId() == null) {
            rate.setId(idGenerator.incrementAndGet());
        }
        rates.put(rate.getBaseCurrency() + "_" + rate.getTargetCurrency(), rate);
        return rate;
    }

    @Override
    public List<CurrencyExchangeRate> findAll() {
        return List.copyOf(rates.values());
    }
}

@Component
class InMemoryFeeConfigurationRepository implements FeeConfigurationRepository {
    private final Map<Long, FeeConfiguration> configs = new ConcurrentHashMap<>();
    private final AtomicLong idGenerator = new AtomicLong();

    public InMemoryFeeConfigurationRepository() {
        configs.put(idGenerator.incrementAndGet(), new FeeConfiguration(null, TransactionType.DEPOSIT, null, BigDecimal.ZERO, null, BigDecimal.ZERO, BigDecimal.ZERO, "No fee for deposits", true));
        configs.put(idGenerator.incrementAndGet(), new FeeConfiguration(null, TransactionType.WITHDRAWAL, null, BigDecimal.ZERO, new BigDecimal("1000.00"), new BigDecimal("1.00"), BigDecimal.ZERO, "Fixed fee for small withdrawals", true));
        configs.put(idGenerator.incrementAndGet(), new FeeConfiguration(null, TransactionType.WITHDRAWAL, null, new BigDecimal("1000.01"), null, BigDecimal.ZERO, new BigDecimal("0.005"), "0.5% for large withdrawals", true));
        configs.put(idGenerator.incrementAndGet(), new FeeConfiguration(null, TransactionType.TRANSFER, null, BigDecimal.ZERO, null, new BigDecimal("0.50"), BigDecimal.ZERO, "Fixed fee for transfers", true));
        configs.put(idGenerator.incrementAndGet(), new FeeConfiguration(null, TransactionType.PAYMENT, null, BigDecimal.ZERO, null, BigDecimal.ZERO, new BigDecimal("0.001"), "0.1% for payments", true));
        configs.put(idGenerator.incrementAndGet(), new FeeConfiguration(null, TransactionType.FX_CONVERSION, null, BigDecimal.ZERO, null, BigDecimal.ZERO, new BigDecimal("0.002"), "0.2% for FX conversions", true));
    }

    @Override
    public List<FeeConfiguration> findByTransactionTypeAndCurrency(TransactionType type, String currency) {
        return configs.values().stream()
                .filter(fc -> fc.isActive() && fc.getTransactionType() == type &&
                        (fc.getCurrency() == null || fc.getCurrency().equalsIgnoreCase(currency)))
                .sorted(Comparator.comparing(FeeConfiguration::getCurrency, Comparator.nullsLast(String::compareTo))) // Currency specific first
                .collect(Collectors.toList());
    }

    @Override
    public List<FeeConfiguration> findByTransactionType(TransactionType type) {
        return configs.values().stream()
                .filter(fc -> fc.isActive() && fc.getTransactionType() == type)
                .collect(Collectors.toList());
    }

    @Override
    public FeeConfiguration save(FeeConfiguration config) {
        if (config.getId() == null) {
            config.setId(idGenerator.incrementAndGet());
        }
        configs.put(config.getId(), config);
        return config;
    }

    @Override
    public Optional<FeeConfiguration> findById(Long id) {
        return Optional.ofNullable(configs.get(id));
    }

    @Override
    public List<FeeConfiguration> findAll() {
        return List.copyOf(configs.values());
    }
}

@Component
class InMemoryScheduledTransactionRepository implements ScheduledTransactionRepository {
    private final Map<Long, ScheduledTransaction> schedules = new ConcurrentHashMap<>();
    private final Map<String, ScheduledTransaction> schedulesById = new ConcurrentHashMap<>();
    private final AtomicLong idGenerator = new AtomicLong();

    public InMemoryScheduledTransactionRepository() {
        // Example: Monthly transfer
        ScheduledTransaction st1 = new ScheduledTransaction(
            "SCHED001", TransactionType.TRANSFER, new BigDecimal("100.00"), "USD",
            1L, 2L, null, "Monthly allowance",
            LocalDateTime.now().plusMinutes(5), "MONTHLY:1", 101L
        );
        st1.setId(idGenerator.incrementAndGet());
        schedules.put(st1.getId(), st1);
        schedulesById.put(st1.getScheduleId(), st1);
    }

    @Override
    public Optional<ScheduledTransaction> findByScheduleId(String scheduleId) {
        return Optional.ofNullable(schedulesById.get(scheduleId));
    }

    @Override
    public ScheduledTransaction save(ScheduledTransaction scheduledTransaction) {
        if (scheduledTransaction.getId() == null) {
            scheduledTransaction.setId(idGenerator.incrementAndGet());
        }
        schedules.put(scheduledTransaction.getId(), scheduledTransaction);
        schedulesById.put(scheduledTransaction.getScheduleId(), scheduledTransaction);
        return scheduledTransaction;
    }

    @Override
    public List<ScheduledTransaction> findByStatusAndNextExecutionTimeBefore(ScheduleStatus status, LocalDateTime dateTime) {
        return schedules.values().stream()
                .filter(st -> st.getStatus() == status && !st.getNextExecutionTime().isAfter(dateTime))
                .collect(Collectors.toList());
    }

    @Override
    public List<ScheduledTransaction> findAll() {
        return List.copyOf(schedules.values());
    }
}
// endregion

// region Core Utilities & Helper Modules

/**
 * Utility for generating IDs.
 */
@Component
```

```
class IdGeneratorService {
    private final AtomicLong nextId = new AtomicLong(1000); // Start IDs higher to avoid collision with initial dummy data

    public Long generateAccountId() { return nextId.incrementAndGet(); }
    public Long generateAccountHolderId() { return nextId.incrementAndGet(); }
    public Long generateTransactionInternalId() { return nextId.incrementAndGet(); }
    public Long generateExchangeRateId() { return nextId.incrementAndGet(); }
    public Long generateFeeConfigId() { return nextId.incrementAndGet(); }
    public Long generateScheduledTransactionId() { return nextId.incrementAndGet(); }
    public String generateSystemTransactionId() { return UUID.randomUUID().toString(); }
    public String generateIdempotencyKey() { return UUID.randomUUID().toString(); }
    public String generateScheduleId() { return "SCHD-" + UUID.randomUUID().toString(); }
    public String generateReportId() { return "RPT-" + UUID.randomUUID().toString(); }
}

/**
 * Utility for BigDecimal operations, ensuring consistent scaling and rounding.
 */
@Component
class CurrencyConverter {
    public BigDecimal scaleAndRound(BigDecimal amount) {
        return amount.setScale(GlobalConfig.CURRENCY_SCALE, GlobalConfig.CURRENCY_ROUNDING_MODE);
    }

    /**
     * Converts an amount from baseCurrency to targetCurrency.
     * @param amount The amount in baseCurrency.
     * @param baseCurrency The currency of the amount.
     * @param targetCurrency The desired currency.
     * @param exchangeRate The rate (1 baseCurrency = X targetCurrency).
     * @return The converted amount in targetCurrency.
     * @throws IllegalArgumentException if currencies are the same but rate is not 1.
     */
    public BigDecimal convert(BigDecimal amount, String baseCurrency, String targetCurrency, BigDecimal exchangeRate) {
        if (baseCurrency.equalsIgnoreCase(targetCurrency)) {
            if (exchangeRate.compareTo(BigDecimal.ONE) != 0) {
                // This scenario should ideally not happen if rates are fetched correctly.
                // For robustness, we enforce a 1:1 conversion if currencies are identical.
                // Or simply return the amount if the goal is only to ensure scaling.
                return scaleAndRound(amount);
            }
            return scaleAndRound(amount);
        }
        return scaleAndRound(amount.multiply(exchangeRate));
    }

    public BigDecimal add(BigDecimal amount1, BigDecimal amount2) {
        return scaleAndRound(amount1.add(amount2));
    }

    public BigDecimal subtract(BigDecimal amount1, BigDecimal amount2) {
        return scaleAndRound(amount1.subtract(amount2));
    }

    public BigDecimal multiply(BigDecimal amount, BigDecimal multiplier) {
        return scaleAndRound(amount.multiply(multiplier));
    }

    public BigDecimal divide(BigDecimal amount, BigDecimal divisor) {
        return scaleAndRound(amount.divide(divisor, GlobalConfig.CURRENCY_ROUNDING_MODE));
    }
}


/**
 * Simple in-memory rate limiter based on client ID.
 */
@Component
class RateLimitingService {
    private static final Logger logger = LoggerFactory.getLogger(RateLimitingService.class);
    // Stores { clientId -> { timestamp of first request in current minute, request count } }
    private final Map<String, Map.Entry<LocalDateTime, AtomicLong>> clientRequestCounts = new ConcurrentHashMap<>();
    private final int limit = GlobalConfig.RATE_LIMIT_PER_MINUTE;
    private final long windowMillis = TimeUnit.MINUTES.toMillis(1);

    public boolean allowRequest(String clientId) {
        if (clientId == null || clientId.isEmpty()) {
            // Treat requests without client ID as allowed or apply a default global limit
            return true;
        }

        LocalDateTime now = LocalDateTime.now();
        clientRequestCounts.compute(clientId, (key, value) -> {
            if (value == null || value.getKey().plusMinutes(1).isBefore(now)) {
                // New minute window or first request
                return Map.entry(now, new AtomicLong(1));
            } else {
                // Same minute window
                value.getValue().incrementAndGet();
                return value;
            }
        });

        long currentCount = clientRequestCounts.get(clientId).getValue().get();
        if (currentCount > limit) {
            logger.warn("Rate limit exceeded for client ID: {} ({} requests)", clientId, currentCount);
            return false;
        }
        return true;
    }

    // A cleanup task for old entries could be added, e.g., via ScheduledExecutorService
}

/**
 * A basic, in-memory event publisher.
 * In a real system, this would be an actual message broker (Kafka, RabbitMQ).
 */
@Component
class EventPublisher {
    private static final Logger logger = LoggerFactory.getLogger(EventPublisher.class);
    private final Map<String, List<Function<Object, Void>>> subscribers = new ConcurrentHashMap<>();

    // Mock an event for transaction completion
    public static class TransactionCompletedEvent {
        public final String systemTransactionId;
        public final TransactionType type;
        public final BigDecimal amount;
        public final String currency;
        public final AccountIdentifier sourceAccount;
        public final AccountIdentifier destinationAccount;
        public final String message;
        public final BigDecimal newBalance; // Primary account balance
        public final String newBalanceCurrency;

        public TransactionCompletedEvent(String systemTransactionId, TransactionType type, BigDecimal amount, String currency, AccountIdentifier sourceAccount, AccountIdentifier destinationAccount, String message, BigDecimal newBalance, String newBalanceCurrency) {
            this.systemTransactionId = systemTransactionId;
            this.type = type;
            this.amount = amount;
```

```
            this.currency = currency;
            this.sourceAccount = sourceAccount;
            this.destinationAccount = destinationAccount;
            this.message = message;
            this.newBalance = newBalance;
            this.newBalanceCurrency = newBalanceCurrency;
        }

        @Override
        public String toString() {
            return "TransactionCompletedEvent{" +
                    "systemTransactionId='" + systemTransactionId + '\'' +
                    ", type=" + type +
                    ", amount=" + amount +
                    ", currency='" + currency + '\'' +
                    '}';
        }
    }

    public static class TransactionFailedEvent {
        public final String systemTransactionId;
        public final TransactionType type;
        public final BigDecimal amount;
        public final String currency;
        public final AccountIdentifier sourceAccount;
        public final AccountIdentifier destinationAccount;
        public final String errorMessage;

        public TransactionFailedEvent(String systemTransactionId, TransactionType type, BigDecimal amount, String currency, AccountIdentifier sourceAccount, AccountIdentifier destinationAccount, String errorMessage) {
            this.systemTransactionId = systemTransactionId;
            this.type = type;
            this.amount = amount;
            this.currency = currency;
            this.sourceAccount = sourceAccount;
            this.destinationAccount = destinationAccount;
            this.errorMessage = errorMessage;
        }
        @Override
        public String toString() {
            return "TransactionFailedEvent{" +
                    "systemTransactionId='" + systemTransactionId + '\'' +
                    ", type=" + type +
                    ", amount=" + amount +
                    ", currency='" + currency + '\'' +
                    ", errorMessage='" + errorMessage + '\'' +
                    '}';
        }
    }

    public <T> void subscribe(String eventType, Function<T, Void> listener) {
        subscribers.computeIfAbsent(eventType, k -> new java.util.ArrayList<>())
                .add(o -> listener.apply((T) o)); // Unchecked cast, typical for generic event systems
        logger.info("Subscribed listener to event type: {}", eventType);
    }

    public void publish(String eventType, Object event) {
        List<Function<Object, Void>> listeners = subscribers.getOrDefault(eventType, Collections.emptyList());
        if (listeners.isEmpty()) {
            logger.debug("No listeners for event type: {}", eventType);
            return;
        }
        logger.debug("Publishing event of type {}: {}", eventType, event);
        listeners.forEach(listener -> {
            try {
                listener.apply(event);
            } catch (Exception e) {
                logger.error("Error processing event for type {}: {}", eventType, e.getMessage(), e);
            }
        });
    }
}

/**
 * Basic in-memory webhook simulator. Subscribes to transaction events.
 */
@Component
class WebhookService {
    private static final Logger logger = LoggerFactory.getLogger(WebhookService.class);

    @Autowired
    public WebhookService(EventPublisher eventPublisher) {
        eventPublisher.subscribe("TransactionCompletedEvent", this::handleTransactionCompleted);
        eventPublisher.subscribe("TransactionFailedEvent", this::handleTransactionFailed);
        logger.info("WebhookService initialized and subscribed to transaction events.");
    }

    private Void handleTransactionCompleted(EventPublisher.TransactionCompletedEvent event) {
        // Simulate sending a webhook notification
        logger.info("WEBHOOK: Sending notification for COMPLETED transaction {}: Type={}, Amount={} {}",
                event.systemTransactionId, event.type, event.amount, event.currency);
        // In a real system, this would involve HTTP calls, retry logic, etc.
        return null;
    }

    private Void handleTransactionFailed(EventPublisher.TransactionFailedEvent event) {
        logger.warn("WEBHOOK: Sending notification for FAILED transaction {}: Type={}, Error={}",
                event.systemTransactionId, event.type, event.errorMessage);
        return null;
    }
}

/**
 * Simplified in-memory Fraud Detection System.
 */
@Component
class FraudDetectionService {
    private static final Logger logger = LoggerFactory.getLogger(FraudDetectionService.class);
    private final TransactionRepository transactionRepository;

    public FraudDetectionService(TransactionRepository transactionRepository) {
        this.transactionRepository = transactionRepository;
    }

    /**
     * Checks if a transaction is potentially fraudulent.
     * This is a very basic rule-based system.
     * In a real system, this would integrate with complex ML models, third-party services.
     *
     * @param transactionId Client provided transaction ID for reference.
     * @param accountId Source or destination account ID.
     * @param amount Transaction amount.
     * @param type Transaction type.
     * @return true if detected as fraudulent, false otherwise.
     */
    public boolean isFraudulent(String transactionId, Long accountId, BigDecimal amount, TransactionType type) {
        // Rule 1: High value transaction
        if (amount.compareTo(GlobalConfig.FRAUD_THRESHOLD_AMOUNT) > 0) {
            logger.warn("Fraud Alert (Rule 1: High Value) for Tx ID: {}, Account: {}, Amount: {}", transactionId, accountId, amount);
            // In a real scenario, this might just flag for review, not block outright.
```

```
            // For simplicity here, we'll return true.
            return true;
        }

        // Rule 2: Excessive daily transactions from an account
        if (accountId != null) {
            LocalDateTime twentyFourHoursAgo = LocalDateTime.now().minusDays(1);
            List<Transaction> recentTransactions = transactionRepository.findByAccountIdAndDateRange(accountId, twentyFourHoursAgo, LocalDateTime.now());
            long dailyCount = recentTransactions.stream()
                                                .filter(tx -> tx.getStatus() == TransactionStatus.COMPLETED)
                                                .count();
            if (dailyCount >= GlobalConfig.MAX_DAILY_TRANSACTIONS) {
                logger.warn("Fraud Alert (Rule 2: High Frequency) for Tx ID: {}, Account: {}, Daily Count: {}", transactionId, accountId, dailyCount);
                return true;
            }
        }

        // Rule 3: Pattern of rapid, small withdrawals (simulated)
        if (type == TransactionType.WITHDRAWAL && accountId != null) {
            LocalDateTime last5Minutes = LocalDateTime.now().minusMinutes(5);
            long rapidWithdrawals = transactionRepository.findByAccountIdAndDateRange(accountId, last5Minutes, LocalDateTime.now())
                                                         .stream()
                                                         .filter(tx -> tx.getType() == TransactionType.WITHDRAWAL && tx.getStatus() == TransactionStatus.COMPLETED)
                                                         .count();
            if (rapidWithdrawals >= 3) { // 3 withdrawals in 5 minutes
                logger.warn("Fraud Alert (Rule 3: Rapid Withdrawals) for Tx ID: {}, Account: {}", transactionId, accountId);
                return true;
            }
        }

        logger.debug("No fraud detected for Tx ID: {}, Account: {}", transactionId, accountId);
        return false;
    }
}

/**
 * Basic Audit Logging Service.
 */
@Component
class AuditService {
    private static final Logger logger = LoggerFactory.getLogger(AuditService.class);

    public void logEvent(String entityType, Long entityId, String eventType, String details, Long userId) {
        logger.info("AUDIT: [{}][ID:{}][{}][User:{}] {}", entityType, entityId, eventType, userId != null ? userId : "N/A", details);
        // In a real system, this would write to a dedicated audit log repository/database, potentially asynchronously.
    }

    public void logTransactionAction(Transaction transaction, String action, String outcome) {
        String details = String.format("TxType: %s, Amount: %s %s, Source: %s, Dest: %s, Status: %s, Outcome: %s",
                transaction.getType(), transaction.getProcessedAmount(), transaction.getProcessedCurrency(),
                transaction.getSourceAccount() != null ? transaction.getSourceAccount().getAccountNumber() : "N/A",
                transaction.getDestinationAccount() != null ? transaction.getDestinationAccount().getAccountNumber() : "N/A",
                transaction.getStatus(), outcome);
        logEvent("TRANSACTION", transaction.getId(), action, details, transaction.getInitiatorUserId());
    }
}

/**
 * Service for managing idempotency keys.
 */
@Component
class IdempotencyService {
    private static final Logger logger = LoggerFactory.getLogger(IdempotencyService.class);
    // Stores { idempotencyKey -> { Transaction, expirationTime } }
    private final Map<String, Map.Entry<Transaction, LocalDateTime>> idempotencyCache = new ConcurrentHashMap<>();

    // A background cleanup task would be beneficial in a real system.
    public Optional<Transaction> getCachedTransaction(String idempotencyKey) {
        Map.Entry<Transaction, LocalDateTime> entry = idempotencyCache.get(idempotencyKey);
        if (entry != null && entry.getValue().isAfter(LocalDateTime.now())) {
            return Optional.of(entry.getKey());
        } else if (entry != null) {
            // Expired entry
            idempotencyCache.remove(idempotencyKey);
        }
        return Optional.empty();
    }

    public void cacheTransaction(String idempotencyKey, Transaction transaction) {
        LocalDateTime expiration = LocalDateTime.now().plusMinutes(GlobalConfig.IDEMPOTENCY_KEY_EXPIRATION_MINUTES);
        idempotencyCache.put(idempotencyKey, Map.entry(transaction, expiration));
        logger.debug("Cached transaction {} for idempotency key {} until {}", transaction.getSystemTransactionId(), idempotencyKey, expiration);
    }
}

/**
 * Service for calculating transaction fees.
 */
@Component
class FeeService {
    private static final Logger logger = LoggerFactory.getLogger(FeeService.class);
    private final FeeConfigurationRepository feeConfigurationRepository;
    private final CurrencyConverter currencyConverter;

    public FeeService(FeeConfigurationRepository feeConfigurationRepository, CurrencyConverter currencyConverter) {
        this.feeConfigurationRepository = feeConfigurationRepository;
        this.currencyConverter = currencyConverter;
    }

    public BigDecimal calculateFees(TransactionType type, BigDecimal amount, String currency) {
        List<FeeConfiguration> configs = feeConfigurationRepository.findByTransactionTypeAndCurrency(type, currency);

        // No specific currency config found, try generic ones
        if (configs.isEmpty() || configs.stream().noneMatch(c -> c.getCurrency() != null && c.getCurrency().equals(currency))) {
            configs.addAll(feeConfigurationRepository.findByTransactionTypeAndCurrency(type, null));
        }

        BigDecimal totalFee = BigDecimal.ZERO;
        boolean feeApplied = false;

        for (FeeConfiguration config : configs) {
            if (!config.isActive()) continue;

            boolean amountMatches = (config.getMinAmount() == null || amount.compareTo(config.getMinAmount()) >= 0) &&
                                    (config.getMaxAmount() == null || amount.compareTo(config.getMaxAmount()) <= 0);

            if (amountMatches) {
                BigDecimal fixed = config.getFixedFee() != null ? config.getFixedFee() : BigDecimal.ZERO;
                BigDecimal percentage = config.getPercentageFee() != null ? amount.multiply(config.getPercentageFee()) : BigDecimal.ZERO;
                totalFee = currencyConverter.add(totalFee, currencyConverter.add(fixed, percentage));
                feeApplied = true;
                logger.debug("Applied fee config (ID: {}) for type {} amount {} {}. Fixed: {}, Percentage: {}", config.getId(), type, amount, currency, fixed, percentage);
            }
        }

        if (!feeApplied && type != TransactionType.DEPOSIT) { // Deposits might legitimately have zero fees
            logger.warn("No active fee configuration found for transaction type {} and amount {} {}", type, amount, currency);
            // Depending on business rules, this might throw an exception or default to zero.
            // For now, we allow zero if no config matches.
```

```
        }

        return currencyConverter.scaleAndRound(totalFee);
    }
}

/**
 * Service for managing currency exchange.
 */
@Component
class CurrencyExchangeService {
    private static final Logger logger = LoggerFactory.getLogger(CurrencyExchangeService.class);
    private final CurrencyExchangeRateRepository exchangeRateRepository;
    private final CurrencyConverter currencyConverter;

    public CurrencyExchangeService(CurrencyExchangeRateRepository exchangeRateRepository, CurrencyConverter currencyConverter) {
        this.exchangeRateRepository = exchangeRateRepository;
        this.currencyConverter = currencyConverter;
    }

    public BigDecimal getExchangeRate(String baseCurrency, String targetCurrency) {
        if (baseCurrency.equalsIgnoreCase(targetCurrency)) {
            return BigDecimal.ONE;
        }

        Optional<CurrencyExchangeRate> rate = exchangeRateRepository.findByBaseCurrencyAndTargetCurrency(baseCurrency, targetCurrency);
        if (rate.isPresent()) {
            return rate.get().getRate();
        }

        // Try inverse rate if available and direct not found
        Optional<CurrencyExchangeRate> inverseRate = exchangeRateRepository.findByBaseCurrencyAndTargetCurrency(targetCurrency, baseCurrency);
        if (inverseRate.isPresent()) {
            // Rate is 1 / inverseRate
            return BigDecimal.ONE.divide(inverseRate.get().getRate(), 8, GlobalConfig.CURRENCY_ROUNDING_MODE); // Higher precision for inverse
        }

        logger.error("Exchange rate not found for {} to {}", baseCurrency, targetCurrency);
        throw new ExchangeRateNotFoundException("Exchange rate not found for " + baseCurrency + " to " + targetCurrency);
    }

    public BigDecimal convertAmount(BigDecimal amount, String sourceCurrency, String targetCurrency) {
        if (sourceCurrency.equalsIgnoreCase(targetCurrency)) {
            return currencyConverter.scaleAndRound(amount);
        }

        BigDecimal rate = getExchangeRate(sourceCurrency, targetCurrency);
        return currencyConverter.convert(amount, sourceCurrency, targetCurrency, rate);
    }
}


/**
 * Service to handle retry logic for external calls or transient failures.
 */
@Component
class RetryService {
    private static final Logger logger = LoggerFactory.getLogger(RetryService.class);

    public interface RetryableAction<T> {
        T execute() throws Exception;
    }

    public <T> T executeWithRetry(RetryableAction<T> action, String actionName) throws Exception {
        int attempts = 0;
        Exception lastException = null;
        while (attempts < GlobalConfig.MAX_RETRIES) {
            try {
                attempts++;
                logger.debug("Attempt {} for action '{}'", attempts, actionName);
                return action.execute();
            } catch (Exception e) {
                lastException = e;
                logger.warn("Attempt {} for action '{}' failed: {}. Retrying in {} seconds...", attempts, actionName, e.getMessage(), GlobalConfig.RETRY_DELAY_SECONDS);
                Thread.sleep(GlobalConfig.RETRY_DELAY_SECONDS * 1000); // Simulate delay
            }
        }
        logger.error("Action '{}' failed after {} attempts.", actionName, GlobalConfig.MAX_RETRIES);
        throw new RuntimeException(actionName + " failed after multiple retries.", lastException);
    }
}

/**
 * Simulates an external payment gateway for processing actual payments.
 */
@Component
class ExternalPaymentGateway {
    private static final Logger logger = LoggerFactory.getLogger(ExternalPaymentGateway.class);
    private final RetryService retryService;

    public ExternalPaymentGateway(RetryService retryService) {
        this.retryService = retryService;
    }

    public String processExternalPayment(String transactionId, BigDecimal amount, String currency, String beneficiaryDetails) throws BadGatewayException {
        // Simulate a call to an external payment gateway
        try {
            return retryService.executeWithRetry(() -> {
                logger.info("Simulating external payment processing for Tx ID: {} to {} amount {} {}", transactionId, beneficiaryDetails, amount, currency);
                // Simulate potential external system failure
                if (Math.random() < 0.1) { // 10% chance of transient failure
                    logger.warn("Simulated transient external gateway failure for Tx ID: {}", transactionId);
                    throw new RuntimeException("External gateway transient error.");
                }
                // Simulate success
                String externalTxRef = "EXT-" + UUID.randomUUID().toString();
                logger.info("External payment for Tx ID: {} completed. External Ref: {}", transactionId, externalTxRef);
                return externalTxRef;
            }, "ExternalPaymentGateway.processExternalPayment");
        } catch (Exception e) {
            logger.error("Failed to process external payment for Tx ID: {} after retries: {}", transactionId, e.getMessage());
            throw new BadGatewayException("Failed to process payment with external gateway: " + e.getMessage(), e);
        }
    }
}

/**
 * Service to manage account locking for concurrent access.
 * This is a highly simplified in-memory lock. In a real distributed system,
 * this would involve distributed locks (e.g., Redlock with Redis).
 */
@Component
class AccountLockingService {
    private static final Logger logger = LoggerFactory.getLogger(AccountLockingService.class);
    private final Set<Long> lockedAccounts = new ConcurrentSkipListSet<>();

    public boolean tryLockAccount(Long accountId) {
        if (lockedAccounts.add(accountId)) {
            logger.debug("Account {} locked.", accountId);
```

```
            return true;
        }
        logger.warn("Account {} is already locked.", accountId);
        return false;
    }

    public void unlockAccount(Long accountId) {
        if (lockedAccounts.remove(accountId)) {
            logger.debug("Account {} unlocked.", accountId);
        } else {
            logger.warn("Attempted to unlock an unlocked account {}.", accountId);
        }
    }
}
// endregion

// region High-Value Business Services

/**
 * Service for reporting and analytics.
 */
@Component
class ReportingService {
    private static final Logger logger = LoggerFactory.getLogger(ReportingService.class);
    private final TransactionRepository transactionRepository;
    private final IdGeneratorService idGeneratorService;

    public ReportingService(TransactionRepository transactionRepository, IdGeneratorService idGeneratorService) {
        this.transactionRepository = transactionRepository;
        this.idGeneratorService = idGeneratorService;
    }

    public TransactionReportResponse generateTransactionReport(TransactionReportRequest request) {
        logger.info("Generating transaction report for account ID: {} from {} to {}", request.getAccountId(), request.getStartDate(), request.getEndDate());

        // Simulate a complex query
        List<Transaction> transactions = transactionRepository.findAll().stream()
                .filter(tx -> request.getAccountId() == null ||
                              (tx.getSourceAccount() != null && tx.getSourceAccount().getAccountId().equals(request.getAccountId())) ||
                              (tx.getDestinationAccount() != null && tx.getDestinationAccount().getAccountId().equals(request.getAccountId())))
                .filter(tx -> request.getStartDate() == null || !tx.getTimestamp().isBefore(request.getStartDate()))
                .filter(tx -> request.getEndDate() == null || !tx.getTimestamp().isAfter(request.getEndDate()))
                .filter(tx -> request.getTypes() == null || request.getTypes().isEmpty() || request.getTypes().contains(tx.getType()))
                .filter(tx -> request.getStatuses() == null || request.getStatuses().isEmpty() || request.getStatuses().contains(tx.getStatus()))
                .sorted(Comparator.comparing(Transaction::getTimestamp))
                .collect(Collectors.toList());

        // Simulate report generation process (e.g., CSV or PDF)
        String reportId = idGeneratorService.generateReportId();
        String downloadLink = "mock_cdn_link/" + reportId + "." + request.getFormat().toLowerCase();
        logger.info("Report {} generated for {} transactions. Format: {}", reportId, transactions.size(), request.getFormat());

        // In a real system, this could be an asynchronous job that emails the link
        return new TransactionReportResponse(reportId, "GENERATED", downloadLink, LocalDateTime.now());
    }

    public BigDecimal getAccountDailySummary(Long accountId, LocalDateTime date) {
        LocalDateTime startOfDay = date.toLocalDate().atStartOfDay();
        LocalDateTime endOfDay = date.toLocalDate().atTime(23, 59, 59, 999999999);

        List<Transaction> dailyTransactions = transactionRepository.findByAccountIdAndDateRange(accountId, startOfDay, endOfDay);

        BigDecimal netChange = BigDecimal.ZERO;
        for (Transaction tx : dailyTransactions) {
            if (tx.getStatus() == TransactionStatus.COMPLETED) {
                if (tx.getDestinationAccount() != null && Objects.equals(tx.getDestinationAccount().getAccountId(), accountId)) {
                    netChange = netChange.add(tx.getProcessedAmount()); // Incoming funds
                }
                if (tx.getSourceAccount() != null && Objects.equals(tx.getSourceAccount().getAccountId(), accountId)) {
                    netChange = netChange.subtract(tx.getProcessedAmount()); // Outgoing funds
                    netChange = netChange.subtract(tx.getFees()); // Fees from source account
                }
            }
        }
        return netChange.setScale(GlobalConfig.CURRENCY_SCALE, GlobalConfig.CURRENCY_ROUNDING_MODE);
    }
}

/**
 * Service for handling scheduled and recurring transactions.
 */
@Component
class ScheduledTransactionProcessorService {
    private static final Logger logger = LoggerFactory.getLogger(ScheduledTransactionProcessorService.class);
    private final ScheduledTransactionRepository scheduledTransactionRepository;
    private final TransactionService transactionService; // To initiate actual transactions
    private final IdGeneratorService idGeneratorService;
    private final AccountRepository accountRepository; // For source/destination accounts (quick lookup)

    // Using a ScheduledExecutorService for background processing
    private final ScheduledExecutorService scheduler = Executors.newSingleThreadScheduledExecutor();

    @Autowired
    public ScheduledTransactionProcessorService(
            ScheduledTransactionRepository scheduledTransactionRepository,
            TransactionService transactionService,
            IdGeneratorService idGeneratorService,
            AccountRepository accountRepository) {
        this.scheduledTransactionRepository = scheduledTransactionRepository;
        this.transactionService = transactionService;
        this.idGeneratorService = idGeneratorService;
        this.accountRepository = accountRepository;

        // Schedule the processing task to run periodically
        scheduler.scheduleAtFixedRate(this::processDueSchedules, 0, GlobalConfig.SCHEDULED_TASK_INTERVAL_MINUTES, TimeUnit.MINUTES);
        logger.info("ScheduledTransactionProcessorService initialized. Processing due schedules every {} minutes.", GlobalConfig.SCHEDULED_TASK_INTERVAL_MINUTES);
    }

    public ScheduledTransaction scheduleTransaction(ScheduledTransactionRequest request) {
        String scheduleId = (request.getScheduleId() != null && !request.getScheduleId().trim().isEmpty())
                            ? request.getScheduleId()
                            : idGeneratorService.generateScheduleId();

        if (scheduledTransactionRepository.findByScheduleId(scheduleId).isPresent()) {
            throw new ScheduledTransactionException("Schedule with ID " + scheduleId + " already exists.");
        }

        // Basic validation
        if (request.getAmount() == null || request.getAmount().compareTo(BigDecimal.ZERO) <= 0) {
            throw new InvalidTransactionException("Scheduled amount must be positive.");
        }
        if (request.getSourceAccountId() == null && request.getType() != TransactionType.DEPOSIT) { // Deposits can have null source, but payment/transfer needs it
            throw new InvalidTransactionException("Source account is required for scheduled " + request.getType());
        }
        if (request.getType() == TransactionType.TRANSFER && request.getDestinationAccountId() == null) {
            throw new InvalidTransactionException("Destination account is required for scheduled transfer.");
        }

        ScheduledTransaction scheduledTx = new ScheduledTransaction(
```

```
            scheduleId, request.getType(), request.getAmount(), request.getCurrency(),
            request.getSourceAccountId(), request.getDestinationAccountId(),
            request.getBeneficiaryDetails(), request.getDescription(),
            request.getScheduledExecutionTime(), request.getRecurrencePattern(),
            request.getCreatedByUserId()
    );

    return scheduledTransactionRepository.save(scheduledTx);
}

@Transactional // Ensure atomicity for scheduled transaction processing
public void processDueSchedules() {
    logger.debug("Scanning for due scheduled transactions...");
    List<ScheduledTransaction> dueSchedules = scheduledTransactionRepository.findByStatusAndNextExecutionTimeBefore(ScheduleStatus.PENDING, LocalDateTime.now());

    if (dueSchedules.isEmpty()) {
        logger.debug("No scheduled transactions due for processing.");
        return;
    }

    logger.info("Found {} scheduled transactions due for processing.", dueSchedules.size());

    for (ScheduledTransaction schedule : dueSchedules) {
        logger.info("Processing scheduled transaction ID: {} (Type: {})", schedule.getScheduleId(), schedule.getType());
        TransactionResponse response = null;
        String resultMessage = "SUCCESS";

        try {
            String idempotencyKey = schedule.getScheduleId() + "-" + schedule.getNextExecutionTime().toLocalDate().toString(); // Ensure idempotency per schedule per day
            switch (schedule.getType()) {
                case DEPOSIT:
                    DepositRequest depositReq = new DepositRequest(
                            idempotencyKey, schedule.getDestinationAccountId(), schedule.getAmount(),
                            schedule.getCurrency(), "Scheduled Deposit: " + schedule.getDescription(),
                            TransactionChannel.BATCH, schedule.getCreatedByUserId(), null
                    );
                    response = transactionService.deposit(depositReq);
                    break;
                case PAYMENT:
                    PaymentRequest paymentReq = new PaymentRequest(
                            idempotencyKey, schedule.getSourceAccountId(), schedule.getAmount(),
                            schedule.getCurrency(), schedule.getBeneficiaryDetails(),
                            "Scheduled Payment: " + schedule.getDescription(),
                            TransactionChannel.BATCH, schedule.getCreatedByUserId(), null
                    );
                    response = transactionService.processPayment(paymentReq);
                    break;
                case TRANSFER:
                    TransferRequest transferReq = new TransferRequest(
                            idempotencyKey, schedule.getSourceAccountId(), schedule.getDestinationAccountId(),
                            schedule.getAmount(), schedule.getCurrency(), "Scheduled Transfer: " + schedule.getDescription(),
                            TransactionChannel.BATCH, schedule.getCreatedByUserId(), null
                    );
                    response = transactionService.transfer(transferReq);
                    break;
                default:
                    throw new InvalidTransactionException("Unsupported transaction type for scheduling: " + schedule.getType());
            }

            if (response.getStatus() != TransactionStatus.COMPLETED) {
                throw new RuntimeException("Transaction not completed: " + response.getMessage());
            }

            schedule.setLastExecutionTime(LocalDateTime.now());
            schedule.setLastExecutionResult(resultMessage);
            schedule.setRetryCount(0); // Reset retry count on success
            // Update next execution time based on recurrence
            updateNextExecutionTime(schedule);
            logger.info("Scheduled transaction {} processed successfully. Next execution: {}", schedule.getScheduleId(), schedule.getNextExecutionTime());

        } catch (Exception e) {
            logger.error("Failed to process scheduled transaction {}: {}", schedule.getScheduleId(), e.getMessage(), e);
            resultMessage = "FAILED: " + e.getMessage();
            schedule.setLastExecutionTime(LocalDateTime.now());
            schedule.setLastExecutionResult(resultMessage);
            schedule.setRetryCount(schedule.getRetryCount() + 1);

            if (schedule.getRetryCount() >= GlobalConfig.MAX_RETRIES) {
                schedule.setStatus(ScheduleStatus.FAILED);
                logger.warn("Scheduled transaction {} permanently failed after {} retries.", schedule.getScheduleId(), GlobalConfig.MAX_RETRIES);
            } else {
                // Implement exponential backoff for next retry attempt, or just a fixed delay
                schedule.setNextExecutionTime(LocalDateTime.now().plusSeconds(GlobalConfig.RETRY_DELAY_SECONDS * schedule.getRetryCount()));
                logger.info("Scheduled transaction {} failed, will retry at {}", schedule.getScheduleId(), schedule.getNextExecutionTime());
            }
        } finally {
            scheduledTransactionRepository.save(schedule); // Save schedule status regardless of success/failure
        }
    }
}

private void updateNextExecutionTime(ScheduledTransaction schedule) {
    LocalDateTime now = LocalDateTime.now();
    switch (schedule.getRecurrencePattern()) {
        case "ONCE":
            schedule.setStatus(ScheduleStatus.COMPLETED); // Mark as completed after one execution
            schedule.setNextExecutionTime(null);
            break;
        case "DAILY":
            schedule.setNextExecutionTime(now.plusDays(1).withHour(schedule.getScheduledExecutionTime().getHour()).withMinute(schedule.getScheduledExecutionTime().getMinute()));
            break;
        case "WEEKLY:MONDAY": // Example for specific day
            LocalDateTime nextWeek = now.plusWeeks(1);
            // Find next Monday
            LocalDateTime nextMonday = nextWeek.with(java.time.DayOfWeek.MONDAY);
            if (nextMonday.isBefore(now)) nextMonday = nextMonday.plusWeeks(1); // If today is Monday after target time, set for next week
            schedule.setNextExecutionTime(nextMonday.withHour(schedule.getScheduledExecutionTime().getHour()).withMinute(schedule.getScheduledExecutionTime().getMinute()));
            break;
        case "MONTHLY:1": // Example for 1st day of month
            LocalDateTime nextMonth = now.plusMonths(1);
            LocalDateTime targetDate = nextMonth.withDayOfMonth(1);
            schedule.setNextExecutionTime(targetDate.withHour(schedule.getScheduledExecutionTime().getHour()).withMinute(schedule.getScheduledExecutionTime().getMinute()));
            break;
        default:
            logger.warn("Unsupported recurrence pattern: {}. Marking schedule {} as completed.", schedule.getRecurrencePattern(), schedule.getScheduleId());
            schedule.setStatus(ScheduleStatus.COMPLETED); // Treat as once if pattern is unknown
            schedule.setNextExecutionTime(null);
            break;
    }
    if (schedule.getNextExecutionTime() != null && schedule.getNextExecutionTime().isBefore(now)) {
        // Adjust for cases where calculation results in a past date, e.g., due to time of day
        schedule.setNextExecutionTime(now.plusDays(1)); // Fallback to next day or proper calculation
    }
}

public void shutdown() {
    logger.info("Shutting down ScheduledTransactionProcessorService scheduler.");
    scheduler.shutdown();
    try {
```

```
            if (!scheduler.awaitTermination(5, TimeUnit.SECONDS)) {
                scheduler.shutdownNow();
            }
        } catch (InterruptedException e) {
            scheduler.shutdownNow();
            Thread.currentThread().interrupt();
        }
    }
}

/**
 * TransactionService: Orchestrates all transaction operations with enhanced features.
 */
@Service
public class TransactionService {

    private static final Logger logger = LoggerFactory.getLogger(TransactionService.class);

    private final AccountRepository accountRepository;
    private final TransactionRepository transactionRepository;
    private final AccountHolderRepository accountHolderRepository;
    private final FraudDetectionService fraudDetectionService;
    private final EventPublisher eventPublisher;
    private final AuditService auditService;
    private final IdempotencyService idempotencyService;
    private final FeeService feeService;
    private final CurrencyExchangeService currencyExchangeService;
    private final CurrencyConverter currencyConverter;
    private final AccountLockingService accountLockingService;
    private final ExternalPaymentGateway externalPaymentGateway;
    private final RateLimitingService rateLimitingService;
    private final IdGeneratorService idGeneratorService;


    @Autowired
    public TransactionService(AccountRepository accountRepository,
                              TransactionRepository transactionRepository,
                              AccountHolderRepository accountHolderRepository,
                              FraudDetectionService fraudDetectionService,
                              EventPublisher eventPublisher,
                              AuditService auditService,
                              IdempotencyService idempotencyService,
                              FeeService feeService,
                              CurrencyExchangeService currencyExchangeService,
                              CurrencyConverter currencyConverter,
                              AccountLockingService accountLockingService,
                              ExternalPaymentGateway externalPaymentGateway,
                              RateLimitingService rateLimitingService,
                              IdGeneratorService idGeneratorService) {
        this.accountRepository = accountRepository;
        this.transactionRepository = transactionRepository;
        this.accountHolderRepository = accountHolderRepository;
        this.fraudDetectionService = fraudDetectionService;
        this.eventPublisher = eventPublisher;
        this.auditService = auditService;
        this.idempotencyService = idempotencyService;
        this.feeService = feeService;
        this.currencyExchangeService = currencyExchangeService;
        this.currencyConverter = currencyConverter;
        this.accountLockingService = accountLockingService;
        this.externalPaymentGateway = externalPaymentGateway;
        this.rateLimitingService = rateLimitingService;
        this.idGeneratorService = idGeneratorService;

        logger.info("TransactionService initialized with enhanced modules.");
    }

    // region Core Transaction Processing Methods (Expanded)

    @Transactional(isolation = Isolation.READ_COMMITTED)
    public TransactionResponse deposit(DepositRequest request) {
        String transactionId = request.getTransactionId();
        if (transactionId == null || transactionId.trim().isEmpty()) {
            transactionId = idGeneratorService.generateIdempotencyKey();
            request.setTransactionId(transactionId);
        }
        String clientId = request.getInitiatorUserId() != null ? String.valueOf(request.getInitiatorUserId()) : "ANONYMOUS";
        if (!rateLimitingService.allowRequest(clientId)) {
            throw new RateLimitExceededException("Too many requests. Please try again later.");
        }

        logger.info("Attempting deposit for account ID: {} with amount: {} {} (Transaction ID: {})",
                request.getAccountId(), request.getAmount(), request.getCurrency(), transactionId);
        auditService.logEvent("DEPOSIT_REQUEST", request.getAccountId(), "INITIATED", "Deposit request received.", request.getInitiatorUserId());

        // 1. Idempotency Check
        Optional<Transaction> existingTx = idempotencyService.getCachedTransaction(transactionId);
        if (existingTx.isPresent()) {
            Transaction tx = existingTx.get();
            if (tx.getStatus() == TransactionStatus.COMPLETED) {
                logger.warn("Transaction with ID {} already processed and completed. Returning existing response.", transactionId);
                auditService.logTransactionAction(tx, "DEPOSIT_COMPLETED", "Idempotent response.");
                return toTransactionResponse(tx);
            } else if (tx.getStatus() == TransactionStatus.FAILED) {
                 // For failed transactions, allow a retry (new attempt with same ID might get processed)
                logger.warn("Transaction with ID {} found but previously FAILED. Allowing re-attempt.", transactionId);
                // Clear cache for failed idempotent transactions to allow re-submission.
                // In a true real-world system, this behavior needs careful consideration and might require a different idempotent key generation strategy for retries.
                idempotencyService.idempotencyCache.remove(transactionId);
            } else {
                logger.warn("Transaction with ID {} found but still PENDING or IN-PROGRESS (Status: {}). Throwing to prevent re-processing.", transactionId, tx.getStatus());
                throw new TransactionAlreadyProcessedException("Transaction with ID " + transactionId + " is already in progress or has a different status. Status: " + tx.getStatus());
            }
        }

        // 2. Validate input amount & currency
        if (request.getAmount() == null || request.getAmount().compareTo(BigDecimal.ZERO) <= 0) {
            logger.error("Invalid deposit amount: {}", request.getAmount());
            throw new InvalidTransactionException("Deposit amount must be positive.");
        }
        String depositCurrency = request.getCurrency() != null ? request.getCurrency().toUpperCase() : GlobalConfig.DEFAULT_CURRENCY;
        request.setCurrency(depositCurrency); // Normalize currency

        // 3. Retrieve target account
        Account account = accountRepository.findById(request.getAccountId())
                .orElseThrow(() -> {
                    logger.error("Account with ID {} not found for deposit.", request.getAccountId());
                    throw new AccountNotFoundException("Account with ID " + request.getAccountId() + " not found.");
                });

        // 4. Currency check
        if (!account.getCurrency().equalsIgnoreCase(depositCurrency)) {
            // Auto-convert if currencies differ
            BigDecimal convertedAmount = currencyExchangeService.convertAmount(request.getAmount(), depositCurrency, account.getCurrency());
            logger.info("Currency conversion for deposit: {} {} -> {} {}. Rate applied.", request.getAmount(), depositCurrency, convertedAmount, account.getCurrency());
            request.setAmount(convertedAmount);
            request.setCurrency(account.getCurrency()); // Update request to reflect target currency and amount
        }
```

```
    // 5. Apply fees for the deposit (if any)
    BigDecimal fees = feeService.calculateFees(TransactionType.DEPOSIT, request.getAmount(), request.getCurrency());
    BigDecimal amountToCredit = currencyConverter.subtract(request.getAmount(), fees);
    if (amountToCredit.compareTo(BigDecimal.ZERO) < 0) {
        throw new FeeConfigurationException("Calculated fees (" + fees + ") exceed the deposit amount (" + request.getAmount() + ").");
    }

    // 6. Fraud check before processing
    if (fraudDetectionService.isFraudulent(transactionId, request.getAccountId(), request.getAmount(), TransactionType.DEPOSIT)) {
        logger.warn("Deposit transaction {} flagged as fraudulent.", transactionId);
        auditService.logEvent("DEPOSIT_FRAUD", request.getAccountId(), "BLOCKED", "Fraud detected.", request.getInitiatorUserId());
        throw new FraudDetectionException("Deposit detected as potentially fraudulent.");
    }

    // 7. Create and save a PENDING transaction record.
    Transaction transaction = new Transaction(
            transactionId,
            TransactionType.DEPOSIT,
            request.getAmount(), // Original amount from request
            request.getCurrency(), // Original currency from request (after initial normalization)
            null, // Source account is null for deposits
            new AccountIdentifier(account.getId(), account.getAccountNumber(), account.getType()),
            request.getDescription(),
            request.getChannel() != null ? request.getChannel() : TransactionChannel.API,
            request.getInitiatorUserId()
    );
    transaction.setFees(fees);
    transaction.setProcessedAmount(amountToCredit);
    transaction.setProcessedCurrency(account.getCurrency());
    transaction.setMetadata(request.getMetadata());
    transaction = transactionRepository.save(transaction); // Persist pending transaction

    // Cache the pending transaction for idempotency
    idempotencyService.cacheTransaction(transactionId, transaction);
    auditService.logTransactionAction(transaction, "DEPOSIT_PENDING", "Transaction recorded as pending.");

    try {
        // 8. Update account balance with optimistic locking
        // Re-fetch account to get latest version for optimistic locking
        Account currentAccount = accountRepository.findById(account.getId()).orElseThrow(() -> new AccountNotFoundException("Account re-fetch failed for ID: " + account.getId()));
        Long expectedVersion = currentAccount.getVersion();

        BigDecimal newBalance = currencyConverter.add(currentAccount.getBalance(), amountToCredit);
        currentAccount.setBalance(newBalance);

        // Validate minimum balance rule only if it's a debit (which deposit is not)
        // But if fees are debited from the account, this might be relevant.
        // For deposits, minimum balance check is not typically applicable for the receiving account.

        accountRepository.saveWithOptimisticLocking(currentAccount, expectedVersion); // Persist updated account balance

        // 9. Mark transaction as COMPLETED
        transaction.setStatus(TransactionStatus.COMPLETED);
        transaction.setSettlementDate(LocalDateTime.now()); // Mark as settled immediately for in-system tx
        transactionRepository.save(transaction);

        eventPublisher.publish("TransactionCompletedEvent", new EventPublisher.TransactionCompletedEvent(
                transaction.getSystemTransactionId(), transaction.getType(), transaction.getProcessedAmount(),
                transaction.getProcessedCurrency(), transaction.getSourceAccount(), transaction.getDestinationAccount(),
                "Deposit successful.", newBalance, account.getCurrency()
        ));

        logger.info("Deposit of {} {} to account {} (ID: {}) completed. New balance: {}",
                transaction.getProcessedAmount(), transaction.getProcessedCurrency(), account.getAccountNumber(), account.getId(), newBalance);
        auditService.logTransactionAction(transaction, "DEPOSIT_COMPLETED", "Deposit successful.");
        return toTransactionResponse(transaction);

    } catch (Exception e) {
        logger.error("Error during deposit for account ID {} (Tx ID: {}): {}", request.getAccountId(), transactionId, e.getMessage(), e);
        transaction.setStatus(TransactionStatus.FAILED);
        transactionRepository.save(transaction); // Update transaction status to FAILED before rollback
        eventPublisher.publish("TransactionFailedEvent", new EventPublisher.TransactionFailedEvent(
                transaction.getSystemTransactionId(), transaction.getType(), transaction.getProcessedAmount(),
                transaction.getProcessedCurrency(), transaction.getSourceAccount(), transaction.getDestinationAccount(),
                e.getMessage()
        ));
        auditService.logTransactionAction(transaction, "DEPOSIT_FAILED", e.getMessage());
        throw e; // Re-throw to trigger full transaction rollback by Spring
    }
}

@Transactional(isolation = Isolation.READ_COMMITTED)
public TransactionResponse processPayment(PaymentRequest request) {
    String transactionId = request.getTransactionId();
    if (transactionId == null || transactionId.trim().isEmpty()) {
        transactionId = idGeneratorService.generateIdempotencyKey();
        request.setTransactionId(transactionId);
    }
    String clientId = request.getInitiatorUserId() != null ? String.valueOf(request.getInitiatorUserId()) : "ANONYMOUS";
    if (!rateLimitingService.allowRequest(clientId)) {
        throw new RateLimitExceededException("Too many requests. Please try again later.");
    }

    logger.info("Attempting payment from account ID: {} with amount: {} {} (Transaction ID: {})",
            request.getSourceAccountId(), request.getAmount(), request.getCurrency(), transactionId);
    auditService.logEvent("PAYMENT_REQUEST", request.getSourceAccountId(), "INITIATED", "Payment request received.", request.getInitiatorUserId());

    // 1. Idempotency Check
    Optional<Transaction> existingTx = idempotencyService.getCachedTransaction(transactionId);
    if (existingTx.isPresent()) {
        Transaction tx = existingTx.get();
        if (tx.getStatus() == TransactionStatus.COMPLETED || tx.getStatus() == TransactionStatus.SETTLED) {
            logger.warn("Transaction with ID {} already processed and completed. Returning existing response.", transactionId);
            auditService.logTransactionAction(tx, "PAYMENT_COMPLETED", "Idempotent response.");
            return toTransactionResponse(tx);
        } else if (tx.getStatus() == TransactionStatus.FAILED) {
            idempotencyService.idempotencyCache.remove(transactionId); // Allow retry
        } else {
            logger.warn("Transaction with ID {} found but still PENDING or IN-PROGRESS (Status: {}). Throwing to prevent re-processing.", transactionId, tx.getStatus());
            throw new TransactionAlreadyProcessedException("Transaction with ID " + transactionId + " is already in progress or has a different status. Status: " + tx.getStatus());
        }
    }

    // 2. Validate input amount & currency
    if (request.getAmount() == null || request.getAmount().compareTo(BigDecimal.ZERO) <= 0) {
        logger.error("Invalid payment amount: {}", request.getAmount());
        throw new InvalidTransactionException("Payment amount must be positive.");
    }
    String paymentCurrency = request.getCurrency() != null ? request.getCurrency().toUpperCase() : GlobalConfig.DEFAULT_CURRENCY;
    request.setCurrency(paymentCurrency);

    // 3. Retrieve source account
    Account sourceAccount = accountRepository.findById(request.getSourceAccountId())
            .orElseThrow(() -> {
                logger.error("Source account with ID {} not found for payment.", request.getSourceAccountId());
                throw new AccountNotFoundException("Source account with ID " + request.getSourceAccountId() + " not found.");
            });
    if (sourceAccount.getStatus() != AccountStatus.ACTIVE) {
        throw new InvalidTransactionException("Source account " + sourceAccount.getAccountNumber() + " is not active.");
```

```
    }

    // 4. Currency conversion if needed for fees/balance check in source account's currency
    BigDecimal processedAmountInSourceCurrency = currencyExchangeService.convertAmount(request.getAmount(), paymentCurrency, sourceAccount.getCurrency());
    BigDecimal fees = feeService.calculateFees(TransactionType.PAYMENT, processedAmountInSourceCurrency, sourceAccount.getCurrency());
    BigDecimal totalDebitAmount = currencyConverter.add(processedAmountInSourceCurrency, fees);

    // 5. Check for sufficient funds and minimum balance
    if (sourceAccount.getBalance().compareTo(totalDebitAmount) < 0) {
        logger.warn("Insufficient funds in account {} (ID: {}) for payment of {}. Current balance: {}", sourceAccount.getAccountNumber(), sourceAccount.getId(), totalDebitAmount, sourceAccount.getBalance());
        throw new InsufficientFundsException("Insufficient funds in source account ID " + request.getSourceAccountId());
    }
    if (currencyConverter.subtract(sourceAccount.getBalance(), totalDebitAmount).compareTo(sourceAccount.getMinimumBalance()) < 0) {
        throw new InsufficientFundsException("Payment would drop account " + sourceAccount.getAccountNumber() + " below minimum balance of " + sourceAccount.getMinimumBalance());
    }

    // 6. Fraud check
    if (fraudDetectionService.isFraudulent(transactionId, request.getSourceAccountId(), request.getAmount(), TransactionType.PAYMENT)) {
        logger.warn("Payment transaction {} flagged as fraudulent.", transactionId);
        auditService.logEvent("PAYMENT_FRAUD", request.getSourceAccountId(), "BLOCKED", "Fraud detected.", request.getInitiatorUserId());
        throw new FraudDetectionException("Payment detected as potentially fraudulent.");
    }

    // 7. Create and save a PENDING transaction record
    Transaction transaction = new Transaction(
            transactionId,
            TransactionType.PAYMENT,
            request.getAmount(),
            paymentCurrency,
            new AccountIdentifier(sourceAccount.getId(), sourceAccount.getAccountNumber(), sourceAccount.getType()),
            new AccountIdentifier(null, request.getBeneficiaryDetails(), AccountType.EXTERNAL), // Destination is external
            request.getDescription() + (request.getBeneficiaryDetails() != null ? " to " + request.getBeneficiaryDetails() : ""),
            request.getChannel() != null ? request.getChannel() : TransactionChannel.API,
            request.getInitiatorUserId()
    );
    transaction.setFees(fees);
    transaction.setProcessedAmount(totalDebitAmount); // This is the amount debited from the source account
    transaction.setProcessedCurrency(sourceAccount.getCurrency());
    transaction.setMetadata(request.getMetadata());
    transaction = transactionRepository.save(transaction);
    idempotencyService.cacheTransaction(transactionId, transaction);
    auditService.logTransactionAction(transaction, "PAYMENT_PENDING", "Transaction recorded as pending.");

    try {
        // 8. Debit source account with optimistic locking
        Account currentSourceAccount = accountRepository.findById(sourceAccount.getId()).orElseThrow(() -> new AccountNotFoundException("Account re-fetch failed for ID: " + sourceAccount.getId()));
        Long expectedVersion = currentSourceAccount.getVersion();

        BigDecimal newBalance = currencyConverter.subtract(currentSourceAccount.getBalance(), totalDebitAmount);
        currentSourceAccount.setBalance(newBalance);
        accountRepository.saveWithOptimisticLocking(currentSourceAccount, expectedVersion);

        // 9. Simulate external payment processing
        String externalRefId = externalPaymentGateway.processExternalPayment(transaction.getSystemTransactionId(), request.getAmount(), paymentCurrency, request.getBeneficiaryDetails());
        transaction.setExternalReferenceId(externalRefId);

        // 10. Mark transaction as COMPLETED and settled
        transaction.setStatus(TransactionStatus.COMPLETED); // Or AUTHORIZED, then CAPTURED/SETTLED
        transaction.setSettlementDate(LocalDateTime.now());
        transactionRepository.save(transaction);

        eventPublisher.publish("TransactionCompletedEvent", new EventPublisher.TransactionCompletedEvent(
                transaction.getSystemTransactionId(), transaction.getType(), transaction.getProcessedAmount(),
                transaction.getProcessedCurrency(), transaction.getSourceAccount(), transaction.getDestinationAccount(),
                "Payment successful.", newBalance, sourceAccount.getCurrency()
        ));

        logger.info("Payment of {} {} from account {} (ID: {}) completed. New balance: {}",
                totalDebitAmount, sourceAccount.getCurrency(), sourceAccount.getAccountNumber(), sourceAccount.getId(), newBalance);
        auditService.logTransactionAction(transaction, "PAYMENT_COMPLETED", "Payment successful, funds debited, external system processed.");
        return toTransactionResponse(transaction);

    } catch (Exception e) {
        logger.error("Error during payment from account ID {} (Tx ID: {}): {}", request.getSourceAccountId(), transactionId, e.getMessage(), e);
        transaction.setStatus(TransactionStatus.FAILED);
        transactionRepository.save(transaction);
        eventPublisher.publish("TransactionFailedEvent", new EventPublisher.TransactionFailedEvent(
                transaction.getSystemTransactionId(), transaction.getType(), transaction.getProcessedAmount(),
                transaction.getProcessedCurrency(), transaction.getSourceAccount(), transaction.getDestinationAccount(),
                e.getMessage()
        ));
        auditService.logTransactionAction(transaction, "PAYMENT_FAILED", e.getMessage());
        throw e;
    }
}

@Transactional(isolation = Isolation.READ_COMMITTED)
public TransactionResponse transfer(TransferRequest request) {
    String transactionId = request.getTransactionId();
    if (transactionId == null || transactionId.trim().isEmpty()) {
        transactionId = idGeneratorService.generateIdempotencyKey();
        request.setTransactionId(transactionId);
    }
    String clientId = request.getInitiatorUserId() != null ? String.valueOf(request.getInitiatorUserId()) : "ANONYMOUS";
    if (!rateLimitingService.allowRequest(clientId)) {
        throw new RateLimitExceededException("Too many requests. Please try again later.");
    }

    logger.info("Attempting transfer from account ID: {} to account ID: {} with amount: {} {} (Transaction ID: {})",
            request.getSourceAccountId(), request.getDestinationAccountId(), request.getAmount(), request.getCurrency(), transactionId);
    auditService.logEvent("TRANSFER_REQUEST", request.getSourceAccountId(), "INITIATED", "Transfer request received.", request.getInitiatorUserId());

    // 1. Idempotency Check
    Optional<Transaction> existingTx = idempotencyService.getCachedTransaction(transactionId);
    if (existingTx.isPresent()) {
        Transaction tx = existingTx.get();
        if (tx.getStatus() == TransactionStatus.COMPLETED) {
            logger.warn("Transaction with ID {} already processed and completed. Returning existing response.", transactionId);
            auditService.logTransactionAction(tx, "TRANSFER_COMPLETED", "Idempotent response.");
            return toTransactionResponse(tx);
        } else if (tx.getStatus() == TransactionStatus.FAILED) {
            idempotencyService.idempotencyCache.remove(transactionId); // Allow retry
        } else {
            logger.warn("Transaction with ID {} found but still PENDING or IN-PROGRESS (Status: {}). Throwing to prevent re-processing.", transactionId, tx.getStatus());
            throw new TransactionAlreadyProcessedException("Transaction with ID " + transactionId + " is already in progress or has a different status. Status: " + tx.getStatus());
        }
    }

    // 2. Validate input and business rules
    if (request.getAmount() == null || request.getAmount().compareTo(BigDecimal.ZERO) <= 0) {
        logger.error("Invalid transfer amount: {}", request.getAmount());
        throw new InvalidTransactionException("Transfer amount must be positive.");
    }
    if (request.getSourceAccountId().equals(request.getDestinationAccountId())) {
        logger.error("Source and destination accounts are the same for transfer from ID: {}", request.getSourceAccountId());
        throw new InvalidTransactionException("Source and destination accounts cannot be the same for a transfer.");
    }
    String transferCurrency = request.getCurrency() != null ? request.getCurrency().toUpperCase() : GlobalConfig.DEFAULT_CURRENCY;
    request.setCurrency(transferCurrency);
```

```
// 3. Retrieve accounts - Fetching order (e.g., by ID) can help prevent deadlocks in a highly concurrent system
// Acquire locks in a consistent order to prevent deadlocks (e.g., always lock lower ID first)
Long firstAccountId = Math.min(request.getSourceAccountId(), request.getDestinationAccountId());
Long secondAccountId = Math.max(request.getSourceAccountId(), request.getDestinationAccountId());

// Attempt to acquire locks for both accounts
if (!accountLockingService.tryLockAccount(firstAccountId)) throw new ConcurrentAccountModificationException("Account " + firstAccountId + " is currently locked.");
try {
    if (!accountLockingService.tryLockAccount(secondAccountId)) throw new ConcurrentAccountModificationException("Account " + secondAccountId + " is currently locked.");
    try {
        Account sourceAccount = accountRepository.findById(request.getSourceAccountId())
                .orElseThrow(() -> {
                    logger.error("Source account with ID {} not found for transfer.", request.getSourceAccountId());
                    return new AccountNotFoundException("Source account with ID " + request.getSourceAccountId() + " not found.");
                });
        Account destinationAccount = accountRepository.findById(request.getDestinationAccountId())
                .orElseThrow(() -> {
                    logger.error("Destination account with ID {} not found for transfer.", request.getDestinationAccountId());
                    return new AccountNotFoundException("Destination account with ID " + request.getDestinationAccountId() + " not found.");
                });

        if (sourceAccount.getStatus() != AccountStatus.ACTIVE) {
            throw new InvalidTransactionException("Source account " + sourceAccount.getAccountNumber() + " is not active.");
        }
        if (destinationAccount.getStatus() != AccountStatus.ACTIVE) {
            throw new InvalidTransactionException("Destination account " + destinationAccount.getAccountNumber() + " is not active.");
        }

        // 4. Determine currency conversion for source/destination if needed
        BigDecimal amountInSourceCurrency = request.getAmount();
        BigDecimal amountInDestinationCurrency = request.getAmount();

        if (!sourceAccount.getCurrency().equalsIgnoreCase(transferCurrency)) {
            amountInSourceCurrency = currencyExchangeService.convertAmount(request.getAmount(), transferCurrency, sourceAccount.getCurrency());
            logger.info("Transfer currency conversion for source: {} {} -> {} {}", request.getAmount(), transferCurrency, amountInSourceCurrency, sourceAccount.getCurrency());
        }
        if (!destinationAccount.getCurrency().equalsIgnoreCase(transferCurrency)) {
            amountInDestinationCurrency = currencyExchangeService.convertAmount(request.getAmount(), transferCurrency, destinationAccount.getCurrency());
            logger.info("Transfer currency conversion for destination: {} {} -> {} {}", request.getAmount(), transferCurrency, amountInDestinationCurrency, destinationAccount.getCurrency());
        } else if (!sourceAccount.getCurrency().equalsIgnoreCase(destinationAccount.getCurrency())) {
            // Direct conversion from source to dest if original is common/implicit
            amountInDestinationCurrency = currencyExchangeService.convertAmount(amountInSourceCurrency, sourceAccount.getCurrency(), destinationAccount.getCurrency());
            logger.info("Direct cross-currency transfer: {} {} -> {} {}", amountInSourceCurrency, sourceAccount.getCurrency(), amountInDestinationCurrency, destinationAccount.getCurrency());
        }

        // 5. Apply fees (typically from source account)
        BigDecimal fees = feeService.calculateFees(TransactionType.TRANSFER, amountInSourceCurrency, sourceAccount.getCurrency());
        BigDecimal totalDebitAmount = currencyConverter.add(amountInSourceCurrency, fees);

        // 6. Check for sufficient funds in the source account and minimum balance
        if (sourceAccount.getBalance().compareTo(totalDebitAmount) < 0) {
            logger.warn("Insufficient funds in source account {} (ID: {}) for transfer of {}. Current balance: {}", sourceAccount.getAccountNumber(), sourceAccount.getId(), totalDebitAmount, sourceAccount.getBalance());
            throw new InsufficientFundsException("Insufficient funds in source account ID " + request.getSourceAccountId());
        }
        if (currencyConverter.subtract(sourceAccount.getBalance(), totalDebitAmount).compareTo(sourceAccount.getMinimumBalance()) < 0) {
            throw new InsufficientFundsException("Transfer would drop account " + sourceAccount.getAccountNumber() + " below minimum balance of " + sourceAccount.getMinimumBalance());
        }

        // 7. Fraud check
        if (fraudDetectionService.isFraudulent(transactionId, request.getSourceAccountId(), request.getAmount(), TransactionType.TRANSFER)) {
            logger.warn("Transfer transaction {} flagged as fraudulent.", transactionId);
            auditService.logEvent("TRANSFER_FRAUD", request.getSourceAccountId(), "BLOCKED", "Fraud detected.", request.getInitiatorUserId());
            throw new FraudDetectionException("Transfer detected as potentially fraudulent.");
        }

        // 8. Create and save a PENDING transaction record
        Transaction transaction = new Transaction(
                transactionId,
                TransactionType.TRANSFER,
                request.getAmount(), // Original request amount
                transferCurrency,
                new AccountIdentifier(sourceAccount.getId(), sourceAccount.getAccountNumber(), sourceAccount.getType()),
                new AccountIdentifier(destinationAccount.getId(), destinationAccount.getAccountNumber(), destinationAccount.getType()),
                request.getDescription(),
                request.getChannel() != null ? request.getChannel() : TransactionChannel.API,
                request.getInitiatorUserId()
        );
        transaction.setFees(fees);
        transaction.setProcessedAmount(totalDebitAmount); // Total amount debited from source
        transaction.setProcessedCurrency(sourceAccount.getCurrency());
        transaction.setMetadata(request.getMetadata());
        // If there was FX conversion, record the exchange rate and converted amounts
        if (!sourceAccount.getCurrency().equalsIgnoreCase(destinationAccount.getCurrency())) {
            BigDecimal rate = currencyExchangeService.getExchangeRate(sourceAccount.getCurrency(), destinationAccount.getCurrency());
            transaction.setExchangeRate(rate);
        }
        transaction = transactionRepository.save(transaction);
        idempotencyService.cacheTransaction(transactionId, transaction);
        auditService.logTransactionAction(transaction, "TRANSFER_PENDING", "Transaction recorded as pending.");

        try {
            // 9. Debit source account with optimistic locking
            Account currentSourceAccount = accountRepository.findById(sourceAccount.getId()).orElseThrow(() -> new AccountNotFoundException("Account re-fetch failed for ID: " + sourceAccount.getId()));
            Long expectedSourceVersion = currentSourceAccount.getVersion();
            currentSourceAccount.setBalance(currencyConverter.subtract(currentSourceAccount.getBalance(), totalDebitAmount));
            accountRepository.saveWithOptimisticLocking(currentSourceAccount, expectedSourceVersion);

            // 10. Credit destination account with optimistic locking
            Account currentDestinationAccount = accountRepository.findById(destinationAccount.getId()).orElseThrow(() -> new AccountNotFoundException("Account re-fetch failed for ID: " + destinationAccount.getId()));
            Long expectedDestinationVersion = currentDestinationAccount.getVersion();
            currentDestinationAccount.setBalance(currencyConverter.add(currentDestinationAccount.getBalance(), amountInDestinationCurrency));
            accountRepository.saveWithOptimisticLocking(currentDestinationAccount, expectedDestinationVersion);

            // 11. Mark transaction as COMPLETED
            transaction.setStatus(TransactionStatus.COMPLETED);
            transaction.setSettlementDate(LocalDateTime.now());
            transactionRepository.save(transaction);

            eventPublisher.publish("TransactionCompletedEvent", new EventPublisher.TransactionCompletedEvent(
                    transaction.getSystemTransactionId(), transaction.getType(), transaction.getProcessedAmount(),
                    transaction.getProcessedCurrency(), transaction.getSourceAccount(), transaction.getDestinationAccount(),
                    "Transfer successful.", currentSourceAccount.getBalance(), currentSourceAccount.getCurrency()
            ));

            logger.info("Transfer of {} {} from account {} (ID: {}) to account {} (ID: {}) completed.",
                    request.getAmount(), transferCurrency, sourceAccount.getAccountNumber(), sourceAccount.getId(), destinationAccount.getAccountNumber(), destinationAccount.getId());
            auditService.logTransactionAction(transaction, "TRANSFER_COMPLETED", "Transfer successful, funds moved.");
            return toTransactionResponse(transaction);

        } catch (Exception e) {
            logger.error("Error during transfer from account ID {} to account ID {} (Tx ID: {}): {}", request.getSourceAccountId(), request.getDestinationAccountId(), transactionId, e.getMessage(), e);
            transaction.setStatus(TransactionStatus.FAILED);
            transactionRepository.save(transaction);
            eventPublisher.publish("TransactionFailedEvent", new EventPublisher.TransactionFailedEvent(
                    transaction.getSystemTransactionId(), transaction.getType(), transaction.getProcessedAmount(),
                    transaction.getProcessedCurrency(), transaction.getSourceAccount(), transaction.getDestinationAccount(),
                    e.getMessage()
            ));
```

```
                auditService.logTransactionAction(transaction, "TRANSFER_FAILED", e.getMessage());
                throw e; // Re-throw to trigger full transaction rollback
            }
        } finally {
            // Ensure locks are released
            accountLockingService.unlockAccount(secondAccountId);
        }
    } finally {
        accountLockingService.unlockAccount(firstAccountId);
    }
}

// endregion

// region Advanced Transaction Operations

/**
 * Processes a refund for a previously completed transaction.
 * This typically credits the source account of the original transaction.
 */
@Transactional(isolation = Isolation.READ_COMMITTED)
public TransactionResponse processRefund(RefundRequest request) {
    String refundTransactionId = request.getTransactionId();
    if (refundTransactionId == null || refundTransactionId.trim().isEmpty()) {
        refundTransactionId = idGeneratorService.generateIdempotencyKey();
        request.setTransactionId(refundTransactionId);
    }
    String clientId = request.getInitiatorUserId() != null ? String.valueOf(request.getInitiatorUserId()) : "ANONYMOUS";
    if (!rateLimitingService.allowRequest(clientId)) {
        throw new RateLimitExceededException("Too many requests. Please try again later.");
    }

    logger.info("Attempting refund for original transaction ID: {} (Refund Tx ID: {})", request.getOriginalTransactionId(), refundTransactionId);
    auditService.logEvent("REFUND_REQUEST", null, "INITIATED", "Refund request received for original Tx ID: " + request.getOriginalTransactionId(), request.getInitiatorUserId());

    // 1. Idempotency Check for the refund itself
    Optional<Transaction> existingRefundTx = idempotencyService.getCachedTransaction(refundTransactionId);
    if (existingRefundTx.isPresent()) {
        Transaction tx = existingRefundTx.get();
        if (tx.getStatus() == TransactionStatus.COMPLETED) {
            logger.warn("Refund transaction with ID {} already processed and completed. Returning existing response.", refundTransactionId);
            return toTransactionResponse(tx);
        } else if (tx.getStatus() == TransactionStatus.FAILED) {
            idempotencyService.idempotencyCache.remove(refundTransactionId);
        } else {
            logger.warn("Refund transaction with ID {} found but still PENDING or IN-PROGRESS (Status: {}). Throwing to prevent re-processing.", refundTransactionId, tx.getStatus());
            throw new TransactionAlreadyProcessedException("Refund transaction with ID " + refundTransactionId + " is already in progress.");
        }
    }

    // 2. Find the original transaction using its idempotency key
    Transaction originalTransaction = transactionRepository.findByTransactionId(request.getOriginalTransactionId())
            .orElseThrow(() -> {
                logger.error("Original transaction with ID {} not found for refund.", request.getOriginalTransactionId());
                throw new AccountNotFoundException("Original transaction with ID " + request.getOriginalTransactionId() + " not found.");
            });

    // 3. Validate original transaction status
    if (originalTransaction.getStatus() != TransactionStatus.COMPLETED && originalTransaction.getStatus() != TransactionStatus.SETTLED) {
        logger.error("Original transaction {} is not in a COMPLETED/SETTLED state for refund. Current status: {}", originalTransaction.getTransactionId(), originalTransaction.getStatus());
        throw new InvalidTransactionException("Original transaction must be completed or settled to be refunded. Current status: " + originalTransaction.getStatus());
    }
    if (originalTransaction.getType() == TransactionType.REFUND || originalTransaction.getType() == TransactionType.REVERSAL) {
         throw new InvalidTransactionException("Cannot refund a refund or reversal transaction.");
    }

    // 4. Determine refund amount
    BigDecimal refundAmount = request.getAmount();
    if (refundAmount == null) {
        refundAmount = originalTransaction.getProcessedAmount(); // Full refund
        // If the original transaction had fees, decide if they are also refunded.
        // For simplicity, we'll refund the processed amount, which might be net of fees.
    } else {
        if (refundAmount.compareTo(BigDecimal.ZERO) <= 0 || refundAmount.compareTo(originalTransaction.getProcessedAmount()) > 0) {
            logger.error("Invalid refund amount: {}. Must be positive and not exceed original processed amount {}.", refundAmount, originalTransaction.getProcessedAmount());
            throw new InvalidTransactionException("Refund amount must be positive and not exceed the original processed amount.");
        }
    }
    refundAmount = currencyConverter.scaleAndRound(refundAmount);

    // 5. Determine the account to be credited (the original source or destination, depending on original tx type)
    AccountIdentifier accountToCreditIdentifier;
    if (originalTransaction.getSourceAccount() != null && originalTransaction.getSourceAccount().getAccountId() != null) {
        // For PAYMENTs or WITHDRAWALs, refund goes back to the source account
        accountToCreditIdentifier = originalTransaction.getSourceAccount();
    } else if (originalTransaction.getDestinationAccount() != null && originalTransaction.getDestinationAccount().getAccountId() != null) {
        // For DEPOSITs, refund means debiting the destination account. This is usually a reversal, not a refund.
        // But if it was a payment from an external source to our account, the refund would go back to that external source,
        // which means debiting our internal account.
        // Let's assume refund for internal accounts means crediting the original sender's account, or for deposits,
        // it means reducing the balance of the recipient account if it was an internal deposit that needs to be reversed.
        // A true "refund" often implies sending money back to an *external* party that paid us.
        // For now, let's treat it as reversing the "effect" on the primary internal account involved.
        // If original was a deposit, it means debiting the account. This is more of a reversal.
        // If original was a payment, it means crediting the source account.
        // Let's assume for simplicity, for any transaction type that led to a debit from an internal account (payment, transfer, withdrawal),
        // a refund means crediting that internal account.
        // If the original was a deposit to an internal account, then a refund means debiting that internal account. This is complex.

        // Simplification: A refund is always a DEBIT from the merchant (our system's internal account)
        // and CREDIT to the original paying account (could be external, or our internal user's account).
        // Here, we assume "refund" means crediting the original source account if it was a debit from them.
        // If it was a deposit to an account, we need to debit that account.
        if (originalTransaction.getType() == TransactionType.DEPOSIT) {
            accountToCreditIdentifier = originalTransaction.getDestinationAccount(); // This is the account that received the deposit, so it should be debited.
                                                                                      // This means the refund will be a 'withdrawal' from their perspective.
        } else { // Transfer, Payment, Withdrawal
            accountToCreditIdentifier = originalTransaction.getSourceAccount(); // This is the account that sent money, it should receive the refund.
        }

        if (accountToCreditIdentifier == null || accountToCreditIdentifier.getAccountId() == null) {
            throw new InvalidTransactionException("Account to credit for refund could not be determined from original transaction.");
        }
    }

    Account creditAccount = accountRepository.findById(accountToCreditIdentifier.getAccountId())
            .orElseThrow(() -> new AccountNotFoundException("Account " + accountToCreditIdentifier.getAccountId() + " not found for refund credit."));

    // 6. Create PENDING refund transaction
    Transaction refundTransaction = new Transaction(
            refundTransactionId,
            TransactionType.REFUND,
            refundAmount,
            originalTransaction.getOriginalCurrency(), // Refund in original currency
            originalTransaction.getSourceAccount(), // Source of original payment
            originalTransaction.getDestinationAccount(), // Destination of original payment
            request.getDescription() != null ? request.getDescription() : "Refund for " + originalTransaction.getTransactionId(),
            request.getChannel() != null ? request.getChannel() : TransactionChannel.API,
            request.getInitiatorUserId()
```

```
    );
    refundTransaction.setParentTransactionId(originalTransaction.getId());
    // For simplicity, no fees on refund
    refundTransaction.setFees(BigDecimal.ZERO);
    refundTransaction.setProcessedAmount(refundAmount);
    refundTransaction.setProcessedCurrency(originalTransaction.getProcessedCurrency());
    refundTransaction = transactionRepository.save(refundTransaction);
    idempotencyService.cacheTransaction(refundTransactionId, refundTransaction);
    auditService.logTransactionAction(refundTransaction, "REFUND_PENDING", "Refund transaction recorded as pending.");

    try {
        // 7. Process the refund: This is essentially a 'deposit' (credit) to the original paying account.
        // The logic below needs to be flexible based on original transaction type and refund nature.
        // For simplicity: If original was a debit from source, refund credits source. If original was credit to dest, refund debits dest.

        Account accountToModify = accountRepository.findById(creditAccount.getId())
                .orElseThrow(() -> new AccountNotFoundException("Account re-fetch failed for ID: " + creditAccount.getId()));
        Long expectedVersion = accountToModify.getVersion();

        BigDecimal newBalance;
        if (originalTransaction.getType() == TransactionType.DEPOSIT) {
            // Refund for a deposit means DEBITING the account that received the deposit
            newBalance = currencyConverter.subtract(accountToModify.getBalance(), refundAmount);
            if (newBalance.compareTo(accountToModify.getMinimumBalance()) < 0) {
                throw new InsufficientFundsException("Refund would drop account " + accountToModify.getAccountNumber() + " below minimum balance.");
            }
        } else {
            // Refund for Payment/Transfer/Withdrawal means CREDITING the account that was debited
            newBalance = currencyConverter.add(accountToModify.getBalance(), refundAmount);
        }

        accountToModify.setBalance(newBalance);
        accountRepository.saveWithOptimisticLocking(accountToModify, expectedVersion);

        // 8. Update original transaction status (e.g., to REFUNDED)
        originalTransaction.setStatus(TransactionStatus.REFUNDED);
        transactionRepository.save(originalTransaction);

        // 9. Mark refund transaction as COMPLETED
        refundTransaction.setStatus(TransactionStatus.COMPLETED);
        refundTransaction.setSettlementDate(LocalDateTime.now());
        transactionRepository.save(refundTransaction);

        eventPublisher.publish("TransactionCompletedEvent", new EventPublisher.TransactionCompletedEvent(
                refundTransaction.getSystemTransactionId(), refundTransaction.getType(), refundTransaction.getProcessedAmount(),
                refundTransaction.getProcessedCurrency(), refundTransaction.getSourceAccount(), refundTransaction.getDestinationAccount(),
                "Refund successful.", newBalance, accountToModify.getCurrency()
        ));

        logger.info("Refund of {} {} for original Tx ID {} completed. New balance for account {}: {}",
                refundAmount, refundTransaction.getProcessedCurrency(), originalTransaction.getTransactionId(), accountToModify.getAccountNumber(), newBalance);
        auditService.logTransactionAction(refundTransaction, "REFUND_COMPLETED", "Refund successful.");
        return toTransactionResponse(refundTransaction);

    } catch (Exception e) {
        logger.error("Error during refund for original Tx ID {} (Refund Tx ID: {}): {}", request.getOriginalTransactionId(), refundTransactionId, e.getMessage(), e);
        refundTransaction.setStatus(TransactionStatus.FAILED);
        transactionRepository.save(refundTransaction);
        eventPublisher.publish("TransactionFailedEvent", new EventPublisher.TransactionFailedEvent(
                refundTransaction.getSystemTransactionId(), refundTransaction.getType(), refundTransaction.getProcessedAmount(),
                refundTransaction.getProcessedCurrency(), refundTransaction.getSourceAccount(), refundTransaction.getDestinationAccount(),
                e.getMessage()
        ));
        auditService.logTransactionAction(refundTransaction, "REFUND_FAILED", e.getMessage());
        throw e;
    }
}

/**
 * Processes a full reversal of a transaction, usually due to an error.
 * Reversals attempt to completely undo the financial impact of the original transaction.
 */
@Transactional(isolation = Isolation.READ_COMMITTED)
public TransactionResponse processReversal(ReversalRequest request) {
    String reversalTransactionId = request.getTransactionId();
    if (reversalTransactionId == null || reversalTransactionId.trim().isEmpty()) {
        reversalTransactionId = idGeneratorService.generateIdempotencyKey();
        request.setTransactionId(reversalTransactionId);
    }
    String clientId = request.getInitiatorUserId() != null ? String.valueOf(request.getInitiatorUserId()) : "ANONYMOUS";
    if (!rateLimitingService.allowRequest(clientId)) {
        throw new RateLimitExceededException("Too many requests. Please try again later.");
    }

    logger.info("Attempting reversal for original transaction ID: {} (Reversal Tx ID: {})", request.getOriginalTransactionId(), reversalTransactionId);
    auditService.logEvent("REVERSAL_REQUEST", null, "INITIATED", "Reversal request received for original Tx ID: " + request.getOriginalTransactionId(), request.getInitiatorUserId());

    // 1. Idempotency Check for the reversal itself
    Optional<Transaction> existingReversalTx = idempotencyService.getCachedTransaction(reversalTransactionId);
    if (existingReversalTx.isPresent()) {
        Transaction tx = existingReversalTx.get();
        if (tx.getStatus() == TransactionStatus.COMPLETED) {
            logger.warn("Reversal transaction with ID {} already processed and completed. Returning existing response.", reversalTransactionId);
            return toTransactionResponse(tx);
        } else if (tx.getStatus() == TransactionStatus.FAILED) {
            idempotencyService.idempotencyCache.remove(reversalTransactionId);
        } else {
            logger.warn("Reversal transaction with ID {} found but still PENDING or IN-PROGRESS (Status: {}). Throwing to prevent re-processing.", reversalTransactionId, tx.getStatus());
            throw new TransactionAlreadyProcessedException("Reversal transaction with ID " + reversalTransactionId + " is already in progress.");
        }
    }

    // 2. Find the original transaction
    Transaction originalTransaction = transactionRepository.findByTransactionId(request.getOriginalTransactionId())
            .orElseThrow(() -> {
                logger.error("Original transaction with ID {} not found for reversal.", request.getOriginalTransactionId());
                throw new AccountNotFoundException("Original transaction with ID " + request.getOriginalTransactionId() + " not found.");
            });

    // 3. Validate original transaction status
    if (originalTransaction.getStatus() != TransactionStatus.COMPLETED && originalTransaction.getStatus() != TransactionStatus.SETTLED) {
        logger.error("Original transaction {} is not in a COMPLETED/SETTLED state for reversal. Current status: {}", originalTransaction.getTransactionId(), originalTransaction.getStatus());
        throw new InvalidTransactionException("Original transaction must be completed or settled to be reversed. Current status: " + originalTransaction.getStatus());
    }
    if (originalTransaction.getStatus() == TransactionStatus.REVERSED || originalTransaction.getStatus() == TransactionStatus.REFUNDED) {
        throw new InvalidTransactionException("Original transaction " + originalTransaction.getTransactionId() + " has already been " + originalTransaction.getStatus().name().toLowerCase() + ".");
    }

    // 4. Create PENDING reversal transaction
    Transaction reversalTransaction = new Transaction(
            reversalTransactionId,
            TransactionType.REVERSAL,
            originalTransaction.getProcessedAmount(), // Reversal amount is the processed amount
            originalTransaction.getProcessedCurrency(),
            originalTransaction.getSourceAccount(),
            originalTransaction.getDestinationAccount(),
            "Reversal for: " + originalTransaction.getSystemTransactionId() + " Reason: " + request.getReason(),
            request.getChannel() != null ? request.getChannel() : TransactionChannel.API,
            request.getInitiatorUserId()
```

```
        );
        reversalTransaction.setFees(originalTransaction.getFees().negate()); // Reverse fees if they were applied
        reversalTransaction.setProcessedAmount(originalTransaction.getProcessedAmount().negate()); // Net effect is opposite
        reversalTransaction.setProcessedCurrency(originalTransaction.getProcessedCurrency());
        reversalTransaction.setParentTransactionId(originalTransaction.getId());
        reversalTransaction = transactionRepository.save(reversalTransaction);
        idempotencyService.cacheTransaction(reversalTransactionId, reversalTransaction);
        auditService.logTransactionAction(reversalTransaction, "REVERSAL_PENDING", "Reversal transaction recorded as pending.");

        try {
            // 5. Reverse the financial impact on accounts
            // This means crediting the original source account and debiting the original destination account
            // or vice-versa, depending on original transaction type.
            // Fees are also reversed.

            AccountIdentifier originalSource = originalTransaction.getSourceAccount();
            AccountIdentifier originalDestination = originalTransaction.getDestinationAccount();
            BigDecimal amountToReverse = originalTransaction.getProcessedAmount(); // Processed amount from original transaction
            BigDecimal feesToReverse = originalTransaction.getFees(); // Fees from original transaction

            BigDecimal newSourceBalance = null; // To return as newBalance
            String newSourceBalanceCurrency = null;

            // Handle source account reversal
            if (originalSource != null && originalSource.getAccountId() != null) {
                Account sourceAccount = accountRepository.findById(originalSource.getAccountId())
                        .orElseThrow(() -> new AccountNotFoundException("Original source account " + originalSource.getAccountId() + " not found for reversal."));

                Account currentSourceAccount = accountRepository.findById(sourceAccount.getId()).orElseThrow(() -> new AccountNotFoundException("Account re-fetch failed for ID: " + sourceAccount.getId()));
                Long expectedSourceVersion = currentSourceAccount.getVersion();

                BigDecimal newBalance = currencyConverter.add(currentSourceAccount.getBalance(), amountToReverse); // Credit back original debit
                if (originalTransaction.getType() == TransactionType.DEPOSIT) { // If original was a deposit to source, debit source
                    newBalance = currencyConverter.subtract(currentSourceAccount.getBalance(), amountToReverse);
                }

                if (originalTransaction.getType() != TransactionType.DEPOSIT) { // Fees were debited from source, so credit them back
                    newBalance = currencyConverter.add(newBalance, feesToReverse);
                }

                // Check minimum balance if it was a debit
                if (originalTransaction.getType() == TransactionType.DEDEBIT || originalTransaction.getType() == TransactionType.PAYMENT || originalTransaction.getType() == TransactionType.TRANSFER) {
                    if (newBalance.compareTo(sourceAccount.getMinimumBalance()) < 0) {
                        throw new InsufficientFundsException("Reversal debit would drop account " + sourceAccount.getAccountNumber() + " below minimum balance.");
                    }
                }

                currentSourceAccount.setBalance(newBalance);
                accountRepository.saveWithOptimisticLocking(currentSourceAccount, expectedSourceVersion);
                newSourceBalance = newBalance;
                newSourceBalanceCurrency = currentSourceAccount.getCurrency();
            }

            // Handle destination account reversal
            if (originalDestination != null && originalDestination.getAccountId() != null &&
                !Objects.equals(originalSource != null ? originalSource.getAccountId() : null, originalDestination.getAccountId())) { // Avoid double-processing same account for self-transfers

                Account destinationAccount = accountRepository.findById(originalDestination.getAccountId())
                        .orElseThrow(() -> new AccountNotFoundException("Original destination account " + originalDestination.getAccountId() + " not found for reversal."));

                Account currentDestinationAccount = accountRepository.findById(destinationAccount.getId()).orElseThrow(() -> new AccountNotFoundException("Account re-fetch failed for ID: " + destinationAccount.getId()));
                Long expectedDestinationVersion = currentDestinationAccount.getVersion();

                BigDecimal newBalance = currencyConverter.subtract(currentDestinationAccount.getBalance(), amountToReverse); // Debit back original credit
                if (originalTransaction.getType() == TransactionType.DEPOSIT) { // If original was a deposit, it means destination was credited, so debit it now
                    newBalance = currencyConverter.subtract(currentDestinationAccount.getBalance(), amountToReverse);
                }

                // Check minimum balance
                if (newBalance.compareTo(destinationAccount.getMinimumBalance()) < 0) {
                    throw new InsufficientFundsException("Reversal debit would drop account " + destinationAccount.getAccountNumber() + " below minimum balance.");
                }

                currentDestinationAccount.setBalance(newBalance);
                accountRepository.saveWithOptimisticLocking(currentDestinationAccount, expectedDestinationVersion);
            }

            // 6. Mark original transaction as REVERSED
            originalTransaction.setStatus(TransactionStatus.REVERSED);
            transactionRepository.save(originalTransaction);

            // 7. Mark reversal transaction as COMPLETED
            reversalTransaction.setStatus(TransactionStatus.COMPLETED);
            reversalTransaction.setSettlementDate(LocalDateTime.now());
            transactionRepository.save(reversalTransaction);

            eventPublisher.publish("TransactionCompletedEvent", new EventPublisher.TransactionCompletedEvent(
                    reversalTransaction.getSystemTransactionId(), reversalTransaction.getType(), reversalTransaction.getProcessedAmount(),
                    reversalTransaction.getProcessedCurrency(), reversalTransaction.getSourceAccount(), reversalTransaction.getDestinationAccount(),
                    "Reversal successful.", newSourceBalance, newSourceBalanceCurrency
            ));

            logger.info("Reversal of original Tx ID {} completed. Reversal Tx ID: {}", originalTransaction.getTransactionId(), reversalTransactionId);
            auditService.logTransactionAction(reversalTransaction, "REVERSAL_COMPLETED", "Reversal successful, original transaction undone.");
            return toTransactionResponse(reversalTransaction);

        } catch (Exception e) {
            logger.error("Error during reversal for original Tx ID {} (Reversal Tx ID: {}): {}", request.getOriginalTransactionId(), reversalTransactionId, e.getMessage(), e);
            reversalTransaction.setStatus(TransactionStatus.FAILED);
            transactionRepository.save(reversalTransaction);
            eventPublisher.publish("TransactionFailedEvent", new EventPublisher.TransactionFailedEvent(
                    reversalTransaction.getSystemTransactionId(), reversalTransaction.getType(), reversalTransaction.getProcessedAmount(),
                    reversalTransaction.getProcessedCurrency(), reversalTransaction.getSourceAccount(), reversalTransaction.getDestinationAccount(),
                    e.getMessage()
            ));
            auditService.logTransactionAction(reversalTransaction, "REVERSAL_FAILED", e.getMessage());
            throw e;
        }
    }

    // endregion

    // region Transaction Query and Reporting

    public List<Transaction> getTransactionsForAccount(TransactionQueryRequest query) {
        if (query.getAccountId() == null) {
            throw new InvalidTransactionException("Account ID is required for transaction query.");
        }
        logger.debug("Fetching transactions for account ID {} with filters: Type={}, Status={}, DateRange={}-{}",
                query.getAccountId(), query.getType(), query.getStatus(), query.getStartDate(), query.getEndDate());
        return transactionRepository.findFilteredTransactions(query);
    }

    public Transaction getTransactionDetails(String transactionId) {
        return transactionRepository.findByTransactionId(transactionId)
                .orElseThrow(() -> new AccountNotFoundException("Transaction with ID " + transactionId + " not found."));
    }

    public TransactionReportResponse requestTransactionReport(TransactionReportRequest request) {
```

```
        // Validate request, e.g., date ranges, account IDs
        if (request.getAccountId() == null) {
            throw new InvalidTransactionException("Account ID is required for reporting.");
        }
        if (request.getStartDate() == null || request.getEndDate() == null) {
            throw new InvalidTransactionException("Start and End dates are required for reporting.");
        }
        if (request.getStartDate().isAfter(request.getEndDate())) {
            throw new InvalidTransactionException("Start date cannot be after end date.");
        }

        // Delegate to ReportingService, which might kick off an async job
        return new ReportingService(transactionRepository, idGeneratorService).generateTransactionReport(request);
    }

    // endregion

    // region Helper Methods

    private TransactionResponse toTransactionResponse(Transaction transaction) {
        // Determine the relevant account balance to return in the response
        BigDecimal newBalance = null;
        String newBalanceCurrency = null;
        if (transaction.getDestinationAccount() != null && transaction.getDestinationAccount().getAccountId() != null && transaction.getType() == TransactionType.DEPOSIT) {
            // For deposit, return destination account's balance
            accountRepository.findById(transaction.getDestinationAccount().getAccountId()).ifPresent(acc -> {
                newBalance = acc.getBalance();
                newBalanceCurrency = acc.getCurrency();
            });
        } else if (transaction.getSourceAccount() != null && transaction.getSourceAccount().getAccountId() != null &&
                (transaction.getType() == TransactionType.PAYMENT || transaction.getType() == TransactionType.TRANSFER)) {
            // For payment/transfer, return source account's balance
            accountRepository.findById(transaction.getSourceAccount().getAccountId()).ifPresent(acc -> {
                newBalance = acc.getBalance();
                newBalanceCurrency = acc.getCurrency();
            });
        } else if (transaction.getType() == TransactionType.REFUND && transaction.getParentTransactionId() != null) {
            // For refund, if it was a credit to source, show source balance
            transactionRepository.findById(transaction.getParentTransactionId()).ifPresent(parentTx -> {
                if (parentTx.getSourceAccount() != null && parentTx.getSourceAccount().getAccountId() != null) {
                    accountRepository.findById(parentTx.getSourceAccount().getAccountId()).ifPresent(acc -> {
                        newBalance = acc.getBalance();
                        newBalanceCurrency = acc.getCurrency();
                    });
                }
            });
        } else if (transaction.getType() == TransactionType.REVERSAL && transaction.getParentTransactionId() != null) {
             transactionRepository.findById(transaction.getParentTransactionId()).ifPresent(parentTx -> {
                if (parentTx.getSourceAccount() != null && parentTx.getSourceAccount().getAccountId() != null) {
                    accountRepository.findById(parentTx.getSourceAccount().getAccountId()).ifPresent(acc -> {
                        newBalance = acc.getBalance();
                        newBalanceCurrency = acc.getCurrency();
                    });
                }
            });
        }

        return new TransactionResponse(
                transaction.getTransactionId(),
                transaction.getSystemTransactionId(),
                transaction.getStatus(),
                transaction.getDescription(),
                newBalance,
                newBalanceCurrency,
                transaction.getFees(),
                transaction.getProcessedAmount(),
                transaction.getProcessedCurrency(),
                transaction.getApprovalStatus()
        );
    }
    // endregion
}
// endregion
```