Complete Core Payment Integration Reference Guide

Analysis Date: June 11, 2025
Documentation Source: Complete Core Payment hierarchy analysis
Primary Page: Core Payment
Target Audience: Developers with limited payment system experience
Purpose: Comprehensive integration reference and educational guide

---

📋 Documentation Structure Overview

This guide is compiled from the **complete Core Payment documentation hierarchy** in the MultiX Engineering space. Here's what was analyzed:

🏠 Main Documentation Tree

```
Core Payment (4256268865) [Main Page - Empty]
├── Core Payment Domain Mapping (4269969235) [Placeholder]
├── How to Integrate with MultiX - Core Payment Platform (4195811595) [Primary Integration Hub]
│   ├── Accessing Current Account Details Guideline (4269982671)
│   ├── Accessing Current Account Transaction Guideline (4269820136) [Empty]
│   ├── Provisioning Mambu Resources Guideline (4270533041)
│   ├── Provisioning Payment Rules/PUC Guideline (4270982093)
│   ├── Integrating Payment Flows with Core Payment Systems Guideline (4270532978)
│   ├── Integrating with Back-office System (Taliad) Guideline (4270533130) [Empty]
│   ├── Register API key for payment-platform-ingress (4415918632)
│   └── MultiX Payment Platform Integration - FAQ (4449665180)
└── How to create a support ticket for the Core Payment Platform (4122642414)

Additional Technical Documentation:
├── [Payment Authorization] API Contract (4195713276) [Complete OpenAPI 3.1.0 Spec]
├── Event Contract - Payment Auth Account Event (4215675545)
├── Event Contract - Payment Order Event (4201809682)
└── [MultiX - API Contract] Current Account (4215180675)
```

🎯 What This Guide Provides

This comprehensive reference will take you from zero knowledge to successful Core Payment integration, covering:

- 📚 **Learning Foundation**: Payment concepts explained for beginners
- 🏗️ **System Architecture**: Understanding all components and their interactions
- ⚙️ **Setup & Configuration**: Complete provisioning and configuration guide

- 🔌 **API Integration**: Step-by-step implementation with real examples
- 📊 **Monitoring & Support**: Operational guidance and troubleshooting
- 🔒 **Security & Compliance**: Authentication, authorization, and best practices

---

🏦 Understanding Core Payment: Foundation Knowledge

What is Core Payment?

Core Payment is the central financial transaction processing engine within the TymeX MultiX ecosystem. It's a sophisticated payment platform that handles all money movement across multiple countries and banking products.

Think of it as:

- The "brain" that validates and processes all financial transactions
- The "gateway" that connects your application to the banking system
- The "orchestrator" that coordinates between different financial services

Why Core Payment Exists

Business Challenge: In a multi-country digital banking environment, you need:

- ✅ Consistent payment processing across countries
- ✅ Regulatory compliance per jurisdiction
- ✅ Secure money movement with proper authorization
- ✅ Real-time balance management
- ✅ Comprehensive audit trails
- ✅ Integration with core banking systems

Core Payment Solution: A unified platform that provides all these capabilities through standardized APIs while handling the complexity behind the scenes.

Core Payment vs. Traditional Banking

Traditional Banking	Core Payment Platform
Manual processes	Automated, API-driven
Country-specific systems	Unified multi-country platform
Batch processing	Real-time processing
Limited integration	Extensive API ecosystem
Rigid workflows	Flexible integration patterns

Geographic Coverage

🇵🇭 GoTyme (Philippines)

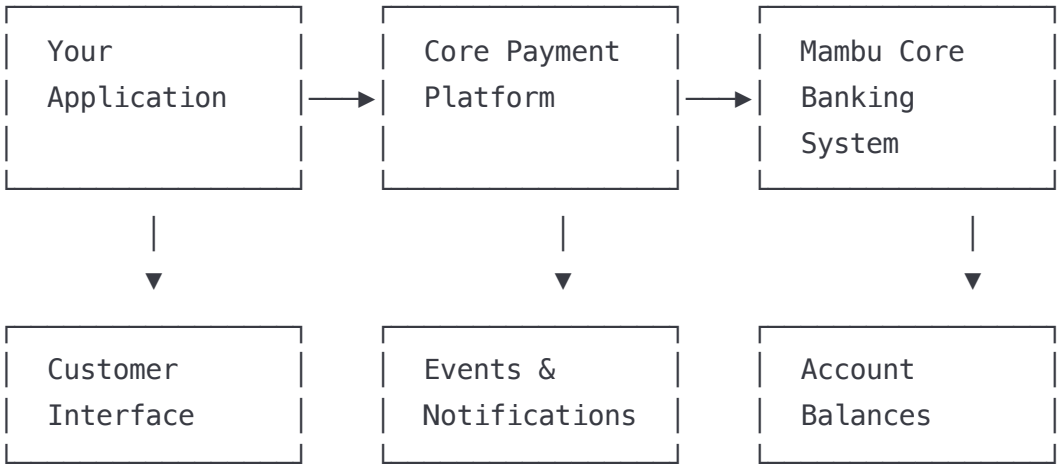
- ✅ **Production**: Fully operational
- ✅ **All Environments**: DEV, TST, STG, PRD available
- ✅ **Full Feature Set**: Complete integration capabilities

🇿🇦 TymeBank/GoSA (South Africa)

- ⚠️ **Limited Availability**: SIT & UAT environments only
- ⚠️ **Testing Restrictions**: Pre-defined account lists only
- 🚧 **Production**: Coming soon

🏗️ System Architecture Deep Dive

The Big Picture: How Money Flows



Core Components Explained

1. Payment Hub Service (Entry Point)

Role: The "front door" that receives all payment requests

Responsibilities:

- 🔐 **Authentication**: Validates API keys and signatures
- ✅ **Input Validation**: Ensures request format compliance
- 🎯 **Routing**: Directs requests to appropriate processing services
- 📊 **Rate Limiting**: Manages request throughput per feature

Location: `payment-platform-ingress` endpoints

2. Core Payment Service (Processing Engine)

Role: The "brain" that orchestrates payment processing

Responsibilities:

- 🔍 **Authorization**: Applies business rules and limits
- 💰 **Balance Management**: Validates account balances
- 📝 **Transaction Processing**: Executes debit/credit operations
- 📤 **Event Publishing**: Publishes results to Kafka topics
- 🔄 **State Management**: Tracks payment status through lifecycle

3. Mambu Core Banking (System of Record)

Role: The "vault" that maintains actual financial records

Responsibilities:

- 💾 **Account Storage**: Customer accounts and balances
- 📋 **Transaction Recording**: Permanent transaction history
- ⚖️ **Regulatory Compliance**: Banking regulation adherence
- 📊 **Reporting**: Financial reporting and statements
- 🔒 **Data Integrity**: Ensures financial data consistency

4. Supporting Services Ecosystem

Notification Engine (Loki Team)

- 📱 SMS notifications for transaction confirmations
- 📧 Email notifications for statements and alerts
- 🔔 Push notifications for mobile applications
- 📨 Notification center for in-app messages

Activity History Service (Neo Team)

- 📈 Transaction history read models
- 🔍 Customer transaction search and filtering
- 📊 Transaction enrichment and categorization
- 📱 Mobile app transaction feeds

Document Engine (Dragon Team)

- 📄 PDF generation for statements and receipts
- 🏦 Bank certificate generation

- 💳 Proof of payment documents
- 📋 Regulatory document creation

Profile Management (Alpha Team)

- 👤 Customer profile data management
- 🔍 Profile search and retrieval APIs
- 📝 Profile updates and lifecycle management
- 🔐 Customer authentication data

Payment Processing Patterns

Core Payment supports **4 distinct integration patterns** based on two key dimensions:

Communication Mechanisms

🔄 Synchronous (Sync)

- Your application sends request and waits for response
- Response time: 2-10 seconds
- Best for: Real-time user interactions
- Use when: User is waiting for immediate confirmation

⚡ Asynchronous (Async)

- Your application sends request and gets immediate acknowledgment
- Final result delivered via Kafka event
- Response time: Immediate acknowledgment, result in 5-30 seconds
- Best for: High-volume or batch processing
- Use when: Can handle eventual consistency

Authorization Methods

📨 SMS (Single Message System)

- Single request completes the entire payment
- Simple, straightforward processing
- Best for: Standard transactions
- Use when: No approval workflow needed

🔄 DMS (Dual Message System)

- Two-step process: Authorize → Confirm/Cancel

- Allows for approval workflows
- Best for: High-value or complex transactions
- Use when: Manual approval or review required

Pattern Combinations & Use Cases

Pattern	Use Cases	Advantages	Considerations
Sync + SMS	• Airtime top-ups • Simple transfers • Bill payments	• Immediate response • Simple implementation • Real-time user feedback	• Application must wait • Limited throughput
Sync + DMS	• Large transfers • Corporate payments • Manual approval workflows	• Control over finalization • Approval workflows • Real-time feedback	• More complex implementation • Multiple API calls required
Async + SMS	• Bulk payments • Payroll processing • High-frequency trading	• High throughput • Non-blocking processing • Scalable	• Eventual consistency • Kafka event handling required
Async + DMS	• Enterprise payments • Multi-step approvals • Complex workflows	• Maximum flexibility • Scalable approval processes • Event-driven architecture	• Most complex implementation • Event ordering considerations

---

⚙️ Complete Setup & Configuration Guide

Phase 1: Understanding Your Payment Flow

Step 1: Business Analysis (CRITICAL FIRST STEP)

⚠️ **WARNING**: Never start coding before completing this step!

Work with your Business Analyst to define:

1. **Accounting Flow Mapping**

```
Example: Bill Payment

Customer Account (Debit: -$102)
     ↓
Utility Company (Credit: +$100) + Fee Account (Credit: +$2)
```

2. **Business Rules Definition**
   - What validations are required?
   - What are the transaction limits?
   - When should payments be rejected?
   - What approval workflows are needed?

3. **Regulatory Requirements**
    - Country-specific compliance rules
    - Reporting requirements
    - Audit trail specifications
    - Data retention policies

Step 2: Payment Use Case (PUC) Design

Payment Use Cases (PUCs) are configuration objects that define how your specific payment type should behave. You need **two types**:

a) Payment Authorization PUC

```yaml
# Example: BillPayment_Authorization_PUC
name: "BillPayment_Auth"
rules:
  - maxAmount: 10000
  - dailyLimit: 50000
  - requiresApproval: false
  - allowedCurrencies: ["PHP", "ZAR"]
validations:
  - accountMustBeActive: true
  - sufficientBalance: true
  - notBlockedCustomer: true
```

b) Payment Processing PUC

```yaml
# Example: BillPayment_Processing_PUC
name: "BillPayment_Processing"
mambuMapping:
  transactionTypes:
    - customerDebit: "BILL_PAYMENT_DEBIT"
    - utilityCredit: "BILL_PAYMENT_CREDIT"
  customFields:
    - billReference: "CUSTOM_FIELD_BILL_REF"
    - utilityProvider: "CUSTOM_FIELD_UTILITY"
```

Phase 2: Infrastructure Provisioning

Step 3: Mambu Resources Setup

What is Mambu? The core banking system that actually holds customer money and account information.

Required Mambu Resources:

a) General Ledger (GL) Accounts

```yaml
# Example GL Account Configuration
GLAccount_UtilityPayments:
  accountCode: "GL-UTILITY-001"
  accountName: "Utility Payments Suspense"
  accountType: "LIABILITY"
  currency: "PHP"
  country: "PH"
```

b) Transaction Channels

```yaml
# Example Transaction Channel
Channel_BillPayment:
  channelId: "BILL_PAY_CHANNEL"
  channelName: "Bill Payment Channel"
  description: "Channel for utility bill payments"
  allowedTransactionTypes: ["DEBIT", "CREDIT"]
```

c) Internal Deposit Accounts (IDA)

```yaml
# Example IDA for Fee Collection
IDA_BillPaymentFees:
  accountId: "IDA-BILLPAY-FEES"
  accountName: "Bill Payment Fee Collection"
  productId: "INTERNAL_ACCOUNT_PRODUCT"
  currency: "PHP"
```

d) Custom Fields (for additional data)

```yaml
# Example Custom Field
CustomField_BillReference:
  fieldId: "BILL_REFERENCE_NUMBER"
  fieldName: "Bill Reference Number"
  dataType: "STRING"
  maxLength: 50
  required: true
```

Step 4: Using the Payment Resource Blueprint

Blueprint Repository: `https://bitbucket.org/tymerepos/tc-payment-resource-blueprint/src/master/`

Repository Selection by Country:

Resource Type	GoTyme (Philippines)	TymeBank (South Africa)
GL Accounts	ph-common-payment-rsc	tb-income-gl-accounts-rsc tb-liability-gl-accounts-rsc tb-expense-gl-accounts-rsc tb-asset-gl-accounts-rsc
Transaction Channels	ph-common-payment-rsc	tb-transaction-channels-rsc
Internal Accounts	Feature team repositories	tb-internal-accounts-rsc
Custom Fields	Feature team repositories	Manually created by Mambu Administrator

Blueprint Usage Process:

1. **Clone** appropriate repository for your country/resource type
2. **Add** your resource definitions to the template
3. **Deploy** using CloudFormation
4. **Test** in lower environments first
5. **Promote** through DEV → TST → STG → PRD

Step 5: API Access Setup

API Key Registration Process:

1. **Determine Your Requirements**

```
Feature: Your application name
Expected RPS: Requests per second needed
Countries: GoTyme, TymeBank, or both
Environments: DEV, TST, STG, PRD
```

2. **Request API Key**
   - Create support ticket using support process
   - Include team name, feature description, and expected load
   - Specify which environments you need access to
3. **Available API Keys**

Feature Type	Expected RPS	SSM Location	Status
Common	50 RPS	`/gotyme/integration/apigw/payment-platform-ingress/common/api-key`	✅ Production Ready
Savings	100 RPS	`/gotyme/integration/apigw/payment-platform-ingress/savings/api-key`	✅ Production Ready
VAS	30 TPS	`/gotyme/integration/apigw/payment-platform-ingress/vas/api-key`	✅ Production Ready
Card Management	20 TPS	`/tp/integration/apigw/payment-platform-ingress/card-mgt/api-key`	✅ Production Ready
Bill Payment	5 TPS	`/tp/integration/apigw/payment-platform-ingress/bill-payment/api-key`	🟡 UAT Ready

Phase 3: Development Environment Setup

Step 6: Environment Configuration

Available Environments:

GoTyme (Philippines)

```bash
DEV:  payment-platform-ingress.dev.tymeph.net
TST:  payment-platform-ingress.tst.tymeph.net
STG:  payment-platform-ingress.stg.tymeph.net
PRD:  payment-platform-ingress.prd.tymeph.net
```

TymeBank (South Africa)

```bash
TST:  payment-platform-ingress.tst.tymesa.net
STG:  payment-platform-ingress.stg.tymesa.net
PRD:  payment-platform-ingress.prd.tymesa.net
```

Step 7: Security Implementation

Authentication Requirements:

a) AWS IAM Signature Version 4 All requests must be signed using AWS Signature V4.

Java Implementation Example:

```java
// Use this library for Java applications
// https://github.com/DarkAtra/feign-aws-sigv4

@Configuration
public class PaymentPlatformConfig {

    @Bean
    public PaymentPlatformClient paymentPlatformClient() {
        return Feign.builder()
            .requestInterceptor(new AWSSignatureV4Interceptor())
            .requestInterceptor(new ApiKeyInterceptor())
            .target(PaymentPlatformClient.class, baseUrl);
    }
}
```

b) Required Headers

```http
Content-Type: application/json
Authorization: AWS4-HMAC-SHA256 Credential=...
x-api-key: YOUR_API_KEY_FROM_SSM
idempotency-key: UNIQUE_REQUEST_IDENTIFIER
caller-id: YOUR_APPLICATION_NAME
```

🔌 Complete API Integration Guide

API Contract Overview

The Core Payment Platform provides **OpenAPI 3.1.0** specification with the following endpoints:

Synchronous Payment APIs

```
POST /internal/payment/auth       # Authorize payment
POST /internal/payment/finalize   # Finalize reserved payment (DMS only)
POST /internal/payment/cancel     # Cancel reserved payment (DMS only)
```

Asynchronous Payment APIs

```
POST /internal/async-payment/auth       # Authorize payment (async)
POST /internal/async-payment/finalize   # Finalize reserved payment (async)
POST /internal/async-payment/cancel     # Cancel reserved payment (async)
```

Current Account APIs

```
GET /internal/account                    # Get account by account number
GET /internal/accounts                   # Get accounts by profile ID
POST /internal/account                   # Create new account
PATCH /internal/account/status           # Update account status
```

Step-by-Step Integration Implementation

Step 8: Request Structure Deep Dive

Core Payment Request Template:

```json
{
  "paymentUseCase": "YourFeature_PaymentType",
  "paymentUseCaseVersion": "1.0.0",
  "paymentDefinition": "YourSpecific_Operation",
  "requestType": "SMS",
  "timestamp": "1718123456789",
  "source": "YourApplicationName",
  "transactions": [
    {
      "transactionDefinition": "CustomerDebit",
      "accountNo": "1234567890",
      "amount": 100.00,
      "currency": "PHP",
      "narrative": "Bill Payment - Electric Company",
      "tags": {
        "BILL_REFERENCE": "INV-2025-001",
        "CUSTOMER_ID": "CUST123"
      }
    }
  ],
  "journalEntries": [
    {
      "journalEntryDefinition": "UtilityPayment",
      "accountNo": "GL-UTILITY-001",
      "amount": 100.00,
      "currency": "PHP"
    }
  ],
  "tags": {
    "PAYMENT_CATEGORY": "UTILITIES",
    "FEATURE_NAME": "BillPay"
  }
}
```

Field Explanations with Examples:

Field	Purpose	Example	Notes
`paymentUseCase`	Links to your configured PUC	`"BillPayment_Utilities"`	Must match exactly
`paymentDefinition`	Specific operation type	`"ElectricBill_Payment"`	Defined in PUC
`requestType`	Processing pattern	`"SMS"` or `"DMS"`	Choose based on needs
`timestamp`	Request creation time	`"1718123456789"`	Unix milliseconds
`transactions`	Account movements	See detailed breakdown below	Array of debits/credits
`journalEntries`	GL account postings	See detailed breakdown below	For accounting
`tags`	Additional metadata	`{"FEATURE": "BillPay"}`	For reporting/tracking

Transaction Object Deep Dive:

```json
{
  "transactionDefinition": "CustomerDebit",
  "accountNo": "1234567890",
  "amount": 100.00,
  "currency": "PHP",
  "narrative": "Bill Payment - Electric Company",
  "tags": {
    "BILL_REFERENCE": "INV-2025-001",
    "CUSTOMER_ID": "CUST123"
  }
}
```

Field	Purpose	Validation	Example
`transactionDefinition`	Links to transaction type in PUC	Must match PUC configuration	`"BillPayment_CustomerDebit"`
`accountNo`	Customer account to debit/credit	5-15 alphanumeric characters	`"51000378543"`
`amount`	Transaction amount	Must be > 0	`100.50`
`currency`	Currency code	3-letter ISO code	`"PHP"`, `"ZAR"`
`narrative`	Description for customer statement	Max 200 characters	`"Electric Bill Payment"`
`tags`	Transaction-specific metadata	Key-value pairs	Additional tracking data

Step 9: Response Handling

Synchronous Success Response (200 OK):

```json
{
  "orderId": "d4bacd57-a083-4cd2-8c60-7a48a0987842",
  "status": "APPROVED",
  "reason": "",
  "authorizedDate": 1734688207253,
  "balances": {
    "51000378543": {
      "accountNo": "51000378543",
      "actualBalance": 862.50,
      "availableBalance": 862.50,
      "reservedDebitBalance": 0,
      "reservedCreditBalance": 0,
      "version": 11
    }
  },
  "transactions": [
    {
      "transactionId": 1831,
      "accountNo": "51000378543",
      "amount": 100,
      "currency": "PHP",
      "transactionDefinition": "BillPayment_CustomerDebit",
      "narrative": "Electric Bill Payment",
      "type": "DEBIT",
      "tags": {
        "BILL_REFERENCE": "INV-2025-001"
      }
    }
  ]
}
```

Asynchronous Acknowledgment Response (202 Accepted):

```json
{
  "asyncMessageId": "async-msg-12345"
}
```

Status Values:

- `APPROVED`: Payment successful (SMS) or reserved (DMS)
- `RESERVED`: Payment authorized, awaiting finalization (DMS only)
- `DECLINED`: Payment rejected

- `DECLINED_RESERVATION`: Authorization failed
- `CANCELLED`: Payment cancelled (DMS only)

Step 10: Error Handling

Error Response Format:

```
json
{
  "errors": [
    {
      "errorCode": "4001001",
      "errorMessage": "Insufficient account balance"
    }
  ]
}
```

Common Error Codes & Solutions:

HTTP Status	Error Code	Error Message	Cause	Solution
400	`4001001`	Insufficient account balance	Not enough money	Check balance before payment
400	`4001002`	Invalid account number	Account doesn't exist	Validate account exists
400	`4001003`	Account is blocked	Account restrictions	Check account status
401	`4010001`	Invalid API key	Wrong or expired key	Verify API key in SSM
401	`4010002`	Invalid signature	AWS signing error	Check signature implementation
404	`4040001`	Payment use case not found	PUC not configured	Verify PUC provisioning
409	`4090001`	Duplicate idempotency key	Key reused	Generate new key for new payments
500	`5000001`	Internal server error	System issue	Contact support via Slack

Step 11: Idempotency Implementation

Critical for Payment Safety:

```javascript
// Correct idempotency key generation
function generateIdempotencyKey(customerId, feature, operation, timestamp) {
  return `${customerId}-${feature}-${operation}-${timestamp}`;
}

// Example usage
const idempotencyKey = generateIdempotencyKey(
  "CUST123",
  "BillPay",
  "ElectricBill",
  Date.now()
);

// Result: "CUST123-BillPay-ElectricBill-1718123456789"
```

Idempotency Rules:

- ✅ **Same key + Same caller = Same result** (safe to retry)
- ❌ **Never reuse keys** across different payments
- ✅ **Include customer ID** to prevent cross-customer issues
- ✅ **Include operation type** to prevent cross-operation issues
- ✅ **Use timestamp** to ensure uniqueness

Step 12: Event-Driven Architecture (Async Patterns)

For Asynchronous Integration, consume events from Kafka:

Payment Order Event Contract:

```json
{
  "eventType": "PAYMENT_COMPLETED",
  "orderId": "d4bacd57-a083-4cd2-8c60-7a48a0987842",
  "status": "APPROVED",
  "paymentUseCase": "BillPayment_Utilities",
  "authorizedDate": 1734688207253,
  "transactions": [...],
  "balances": {...}
}
```

Event Topics:

- `payment-order-events`: Payment completion results

- `payment-auth-account-events`: Account balance changes
- `bank-account-events`: Account lifecycle events

Event Consumer Implementation:

```java
@KafkaListener(topics = "payment-order-events")
public void handlePaymentOrderEvent(PaymentOrderEvent event) {
    if ("APPROVED".equals(event.getStatus())) {
        // Handle successful payment
        sendConfirmationToCustomer(event.getOrderId());
        updateApplicationState(event);
    } else {
        // Handle failed payment
        notifyCustomerOfFailure(event.getOrderId(), event.getReason());
    }
}
```

💼 Real-World Integration Examples

Example 1: Complete Bill Payment Integration

Business Scenario:

- Customer pays utility bill through mobile app
- $2 convenience fee charged
- SMS confirmation sent
- Transaction recorded in activity history

Step-by-Step Implementation

1. Business Flow Definition:

```
Customer Account: -$102 (bill amount + fee)
Utility GL Account: +$100 (bill payment)
Fee Collection Account: +$2 (convenience fee)
```

2. PUC Configuration:

yaml

```yaml
# Authorization PUC
PaymentAuthPUC:
  name: "BillPayment_Utility_Auth"
  limits:
    maxAmount: 50000
    dailyLimit: 200000
  validations:
    - sufficientBalance: true
    - accountActive: true
    - notBlockedCustomer: true

# Processing PUC
PaymentProcessingPUC:
  name: "BillPayment_Utility_Processing"
  transactions:
    - customerDebit: "BILL_PAYMENT_CUSTOMER_DEBIT"
    - utilityCredit: "BILL_PAYMENT_UTILITY_CREDIT"
  customFields:
    - billReference: "BILL_REF_NUMBER"
    - utilityProvider: "UTILITY_PROVIDER_CODE"
```

3. API Implementation:

javascript

```
class BillPaymentService {

  async processBillPayment(customerId, billAmount, utilityAccount, billReference) {
    const totalAmount = billAmount + 2.00; // Add convenience fee

    const paymentRequest = {
      paymentUseCase: "BillPayment_Utility",
      paymentUseCaseVersion: "1.0.0",
      paymentDefinition: "UtilityBillPayment",
      requestType: "SMS",
      timestamp: Date.now().toString(),
      source: "MobileBillPayApp",
      transactions: [
        {
          transactionDefinition: "CustomerDebit",
          accountNo: customerId,
          amount: totalAmount,
          currency: "PHP",
          narrative: `Utility Bill Payment - ${utilityAccount}`,
          tags: {
            BILL_REFERENCE: billReference,
            BILL_AMOUNT: billAmount.toString(),
            CONVENIENCE_FEE: "2.00"
          }
        }
      ],
      journalEntries: [
        {
          journalEntryDefinition: "UtilityPayment",
          accountNo: "GL-UTILITY-PAYMENTS",
          amount: billAmount,
          currency: "PHP"
        },
        {
          journalEntryDefinition: "FeeCollection",
          accountNo: "GL-CONVENIENCE-FEES",
          amount: 2.00,
          currency: "PHP"
        }
      ],
      tags: {
        BILL_TYPE: "UTILITY",
        CUSTOMER_ID: customerId,
        UTILITY_PROVIDER: utilityAccount
      }
    };
```

```
  try {
    const response = await this.callCorePaymentAPI(paymentRequest);

    if (response.status === 'APPROVED') {
      await this.sendSMSConfirmation(customerId, response.orderId, totalAmount);
      await this.recordActivityHistory(customerId, response);
      return { success: true, orderId: response.orderId };
    } else {
      return { success: false, error: response.reason };
    }

  } catch (error) {
    console.error('Payment failed:', error);
    return { success: false, error: 'System error occurred' };
  }
}

async callCorePaymentAPI(paymentRequest) {
  const idempotencyKey = this.generateIdempotencyKey(
    paymentRequest.tags.CUSTOMER_ID,
    'BillPay',
    paymentRequest.tags.BILL_REFERENCE
  );

  const response = await fetch(
    'https://payment-platform-ingress.prd.tymeph.net/internal/payment/auth',
    {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json',
        'x-api-key': process.env.CORE_PAYMENT_API_KEY,
        'idempotency-key': idempotencyKey,
        'caller-id': 'BillPaymentService',
        'Authorization': this.generateAWSSignature(paymentRequest)
      },
      body: JSON.stringify(paymentRequest)
    }
  );

  return await response.json();
}

generateIdempotencyKey(customerId, feature, billReference) {
  return `${customerId}-${feature}-${billReference}-${Date.now()}`;
```

```
    }
}
```

Example 2: High-Volume Async Processing

Business Scenario:

- Payroll disbursement to 10,000 employees
- Batch processing with eventual consistency
- Progress tracking via events

Implementation:

javascript

```
class PayrollService {

  async processPayrollBatch(employees) {
    const batchId = `PAYROLL-${Date.now()}`;
    const results = [];

    for (const employee of employees) {
      const paymentRequest = {
        paymentUseCase: "Payroll_Disbursement",
        paymentUseCaseVersion: "1.0.0",
        paymentDefinition: "SalaryPayment",
        requestType: "SMS",
        timestamp: Date.now().toString(),
        source: "PayrollSystem",
        transactions: [
          {
            transactionDefinition: "EmployeeCredit",
            accountNo: employee.accountNo,
            amount: employee.salary,
            currency: "PHP",
            narrative: `Salary Payment - ${employee.employeeId}`,
            tags: {
              EMPLOYEE_ID: employee.employeeId,
              PAYROLL_BATCH: batchId,
              SALARY_PERIOD: "2025-06"
            }
          }
        ],
        journalEntries: [
          {
            journalEntryDefinition: "PayrollExpense",
            accountNo: "GL-PAYROLL-EXPENSE",
            amount: employee.salary,
            currency: "PHP"
          }
        ],
        tags: {
          BATCH_ID: batchId,
          EMPLOYEE_COUNT: employees.length.toString(),
          TOTAL_AMOUNT: this.calculateTotalAmount(employees).toString()
        }
      };

      try {
        // Use async endpoint for high volume
        const response = await this.callAsyncPaymentAPI(paymentRequest);
```

```
      results.push({
        employeeId: employee.employeeId,
        asyncMessageId: response.asyncMessageId,
        status: 'SUBMITTED'
      });
    } catch (error) {
      results.push({
        employeeId: employee.employeeId,
        status: 'FAILED',
        error: error.message
      });
    }
  }

  return {
    batchId,
    totalSubmitted: results.filter(r => r.status === 'SUBMITTED').length,
    totalFailed: results.filter(r => r.status === 'FAILED').length,
    results
  };
}

async callAsyncPaymentAPI(paymentRequest) {
  const response = await fetch(
    'https://payment-platform-ingress.prd.tymeph.net/internal/async-payment/auth?pri
    {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json',
        'x-api-key': process.env.CORE_PAYMENT_API_KEY,
        'idempotency-key': this.generateIdempotencyKey(paymentRequest),
        'caller-id': 'PayrollService',
        'Authorization': this.generateAWSSignature(paymentRequest)
      },
      body: JSON.stringify(paymentRequest)
    }
  );

  return await response.json();
}

// Event consumer for async results
@KafkaListener(topics = "payment-order-events")
handlePayrollResult(event) {
  if (event.tags && event.tags.BATCH_ID) {
    this.updatePayrollStatus(
      event.tags.BATCH_ID,
```

```
        event.tags.EMPLOYEE_ID,
        event.status
      );

      if (event.status === 'APPROVED') {
        this.notifyEmployee(event.tags.EMPLOYEE_ID, event.orderId);
      }
    }
  }
}
```

🔧 Account Management Integration

Current Account APIs

The Core Payment platform also provides comprehensive account management capabilities:

Get Account Information

API: `GET /internal/account?accountNo={accountNumber}`

Use Cases:

- Validate account exists before payment
- Check account status and balances
- Retrieve account details for customer service

Request Example:

bash

```
curl -X GET \
  "https://payment-platform-ingress.prd.tymeph.net/internal/account?accountNo=51000378
  -H "x-api-key: YOUR_API_KEY" \
  -H "Authorization: AWS4-HMAC-SHA256..."
```

Response Example:

```json
{
  "accountNo": "51000378543",
  "profileId": "PROF-123456",
  "productId": "EVERYDAY_DEPOSIT",
  "accountName": "John Doe - Current Account",
  "currency": "PHP",
  "status": "ACTIVE",
  "balance": {
    "actualBalance": 1250.75,
    "availableBalance": 1250.75,
    "reservedDebitBalance": 0.00,
    "reservedCreditBalance": 0.00
  },
  "createdDate": "2025-01-15T10:30:00Z",
  "lastModifiedDate": "2025-06-11T14:45:00Z"
}
```

Create New Account

API: `POST /internal/account`

Use Cases:

- Onboarding new customers
- Opening additional accounts for existing customers
- Creating specialized accounts (savings, etc.)

Request Example:

```json
{
  "profileId": "PROF-123456",
  "productId": "EVERYDAY_DEPOSIT",
  "accountName": "John Doe - Savings Account",
  "currency": "PHP",
  "initialDeposit": 1000.00,
  "accountHolders": [
    {
      "profileId": "PROF-123456",
      "role": "PRIMARY_HOLDER"
    }
  ]
}
```

Update Account Status

API: `PATCH /internal/account/status`

Use Cases:

- Block/unblock accounts for security
- Apply regulatory holds
- Manage account lifecycle

Request Example:

```
json

{
  "accountNo": "51000378543",
  "status": "BLOCKED",
  "reasonCode": "SUSPECTED_FRAUD",
  "comment": "Unusual transaction pattern detected",
  "holdCodes": ["FRAUD_HOLD"]
}
```

Account Events Integration

Subscribe to account-related events for real-time updates:

Bank Account Event Example:

```
json

{
  "eventType": "ACCOUNT_STATUS_CHANGED",
  "accountNo": "51000378543",
  "profileId": "PROF-123456",
  "oldStatus": "ACTIVE",
  "newStatus": "BLOCKED",
  "timestamp": "2025-06-11T14:45:00Z",
  "reasonCode": "SUSPECTED_FRAUD"
}
```

Payment Auth Account Event Example:

```json
{
  "eventType": "ACCOUNT_BALANCE_CHANGED",
  "accountNo": "51000378543",
  "previousBalance": 1250.75,
  "currentBalance": 1150.75,
  "changeAmount": -100.00,
  "changeType": "PAYMENT_DEBIT",
  "orderId": "d4bacd57-a083-4cd2-8c60-7a48a0987842",
  "timestamp": "2025-06-11T14:45:00Z"
}
```

🚨 Troubleshooting & Common Issues

Authentication & Authorization Issues

Problem: 401 Unauthorized - Invalid API Key

Symptoms:

- HTTP 401 responses
- Error message: "Invalid API key"

Solutions:

1. **Verify API Key Source:**

   ```bash
   # Check SSM parameter
   aws ssm get-parameter --name "/gotyme/integration/apigw/payment-platform-ingress/co
   ```

2. **Validate Header Format:**

   ```http
   x-api-key: YOUR_ACTUAL_API_KEY_HERE
   # NOT: x-api-key: "YOUR_API_KEY" (no quotes)
   # NOT: X-API-Key: YOUR_API_KEY (wrong case)
   ```

3. **Check Environment:**
   - DEV environment key won't work in PROD
   - Ensure you're using the correct environment

Problem: 401 Unauthorized - Invalid AWS Signature

Symptoms:

- HTTP 401 responses
- Error message: "The request signature we calculated does not match the signature you provided"

Solutions:

1. **Verify System Clock:**

```bash
# AWS requires accurate time (within 5 minutes)
sudo ntpdate -s time.nist.gov
```

2. **Check Signature Implementation:**

```java
// Correct Java implementation
@Bean
public RequestInterceptor awsSignatureInterceptor() {
    return new AWSV4SignerInterceptor(
        "execute-api",  // Service name
        awsRegion,      // Region
        awsCredentials  // Credentials
    );
}
```

3. **Validate Request Canonicalization:**
   - Ensure headers are lowercase
   - Include required headers in signature
   - URL encode path parameters correctly

Payment Processing Issues

Problem: 400 Bad Request - Insufficient Funds

Symptoms:

- HTTP 400 response
- Error code: "4001001"
- Error message: "Insufficient account balance"

Solutions:

1. **Check Account Balance:**

```javascript
// Pre-validate balance before payment
async function validateBalance(accountNo, amount) {
  const account = await getAccountDetails(accountNo);
  return account.balance.availableBalance >= amount;
}
```

2. **Consider Reserved Amounts:**
   - Account might have pending transactions
   - Check `reservedDebitBalance` in account response
   - Available balance = actual balance - reserved amounts
3. **Handle Gracefully:**

```javascript
if (error.errorCode === '4001001') {
  return {
    success: false,
    errorType: 'INSUFFICIENT_FUNDS',
    message: 'Please ensure sufficient balance and try again',
    availableBalance: account.balance.availableBalance
  };
}
```

Problem: 404 Not Found - Payment Use Case Not Found

Symptoms:

- HTTP 404 response
- Error code: "4040001"
- Error message: "Payment use case not found"

Solutions:

1. **Verify PUC Deployment:**

```bash
# Check if PUC exists in target environment
# Contact Core Payment team to verify deployment status
```

2. **Check PUC Name Accuracy:**

```javascript
// Ensure exact match with deployed PUC
const paymentUseCase = "BillPayment_Utility"; // Must match exactly
```

3. **Validate Environment:**
   - PUC might be deployed in PROD but not TST
   - Check with Core Payment team for environment status

Problem: 409 Conflict - Duplicate Idempotency Key

Symptoms:

- HTTP 409 response
- Error code: "4090001"
- Error message: "Duplicate idempotency key"

Solutions:

1. **Check Key Generation Logic:**

```javascript
// WRONG: Reusing the same key
const key = `${customerId}-payment`;

// CORRECT: Include unique elements
const key = `${customerId}-${operation}-${timestamp}-${uuid()}`;
```

2. **Understand Idempotency Scope:**
   - Same key + same caller-id = same result
   - Different caller-id can use same key
   - Keys are scoped per API endpoint

3. **Handle Duplicate Gracefully:**

```javascript
if (error.errorCode === '4090001') {
  // This might be a legitimate retry
  // Return the cached result if available
  return await getCachedPaymentResult(idempotencyKey);
}
```

Network & Performance Issues

Problem: Timeouts and Slow Responses

Symptoms:

- Requests taking > 30 seconds
- Connection timeout errors
- HTTP 504 Gateway Timeout

Solutions:

1. **Implement Proper Timeouts:**

```javascript
const controller = new AbortController();
const timeoutId = setTimeout(() => controller.abort(), 15000); // 15 second timeout

try {
  const response = await fetch(url, {
    signal: controller.signal,
    ...options
  });
} catch (error) {
  if (error.name === 'AbortError') {
    // Handle timeout
  }
} finally {
  clearTimeout(timeoutId);
}
```

2. **Use Async APIs for High Volume:**

```javascript
// For high-volume or batch processing
// Use async endpoints instead of sync
const asyncResponse = await fetch('/internal/async-payment/auth');
```

3. **Implement Exponential Backoff:**

```javascript
async function retryWithBackoff(fn, maxRetries = 3) {
  for (let i = 0; i < maxRetries; i++) {
    try {
      return await fn();
    } catch (error) {
      if (i === maxRetries - 1) throw error;

      const delay = Math.pow(2, i) * 1000; // 1s, 2s, 4s
      await new Promise(resolve => setTimeout(resolve, delay));
    }
  }
}
```

Problem: Rate Limiting (HTTP 429)

Symptoms:

- HTTP 429 Too Many Requests
- Error message: "Rate limit exceeded"

Solutions:

1. **Check Your API Key Limits:**

```javascript
// Review your registered RPS/TPS limits
// Contact Core Payment team to increase if needed
```

2. **Implement Rate Limiting:**

```javascript
const RateLimiter = require('limiter').RateLimiter;
const limiter = new RateLimiter(50, 'second'); // 50 requests per second

function makeRequest() {
  return new Promise((resolve, reject) => {
    limiter.removeTokens(1, (err, remainingRequests) => {
      if (err) reject(err);
      else resolve(callPaymentAPI());
    });
  });
}
```

3. **Use Async Processing:**

```javascript
// For high-volume scenarios
// Batch requests using async endpoints
// Process results via Kafka events
```

Event Processing Issues

Problem: Missing Kafka Events

Symptoms:

- Async payments submitted but no completion events received
- Event consumer not receiving messages

Solutions:

1. **Verify Topic Subscription:**

```java
@KafkaListener(
  topics = "payment-order-events",
  groupId = "your-service-consumer-group"
)
public void handlePaymentEvent(PaymentOrderEvent event) {
  // Process event
}
```

2. **Check Consumer Group Configuration:**

```properties
# application.properties
spring.kafka.consumer.group-id=your-unique-group-id
spring.kafka.consumer.auto-offset-reset=earliest
spring.kafka.consumer.enable-auto-commit=true
```

3. **Handle Event Ordering:**

```java
// Events might arrive out of order
// Use orderId to correlate events
// Store state and handle retries

if (event.getOrderId().equals(expectedOrderId)) {
  processEvent(event);
} else {
  // Handle out-of-order event
  storeForLaterProcessing(event);
}
```

Environment-Specific Issues

Problem: TymeBank Testing Limitations

Symptoms:

- Payments fail in TymeBank environments
- Account not found errors

Solutions:

1. **Use Pre-defined Test Accounts:**

```javascript
// TymeBank has limited test accounts
// Contact Core Payment team for account list
const testAccounts = [
  "TB-TEST-001",
  "TB-TEST-002"
  // Get complete list from Core Payment team
];
```

2. **Check Environment Availability:**

```javascript
// TymeBank only available in:
// - SIT (System Integration Testing)
// - UAT (User Acceptance Testing)
// Production not yet available
```

3. **Validate Account Existence:**

```javascript
// Always check if account exists before payment
const account = await getAccountDetails(accountNo);
if (!account) {
  throw new Error('Account not available in this environment');
}
```

📞 Support & Resources

Getting Help

Slack Support Channels (Real-time Help)

Primary Support Channel:

- **#tx-core-payment-hub**: Main support channel for Core Payment issues
  - Tag: @corepayment-team for urgent issues
  - Response time: 1-4 hours during business hours
  - Escalation available for production issues

Related Support Channels:

- **#notification-engine-forum**: SMS/email notification issues
- **#multix-profile-management-forum**: Customer profile questions
- **#multix-current-account-forum**: Account management issues
- **#step-up-forum**: Authentication and security questions

Creating Support Tickets

When to Create Tickets:

- API key requests
- PUC provisioning help
- Production issues
- Feature requests
- Environment access requests

Required Information:

```markdown
**Title:** [Portfolio Name] - [Country] - [Issue Description]
**Portfolio:** Your application/feature name
**Countries:** GoTyme, TymeBank, or both
**Environment:** DEV, TST, STG, PRD
**Issue Type:** [Integration Help, Production Issue, Feature Request]
**Description:**
- What you were trying to do
- What actually happened
- Error messages (if any)
- Steps to reproduce
**Expected Outcome:** What should happen
**Business Impact:** Why this needs to be resolved
**Due Date:** When you need this resolved
```

Support Process:

1. **Create Jira Ticket** under Epic: CPP-130
2. **Post to Slack** #tx-core-payment-hub with ticket link
3. **Tag Relevant People:**
   - Core Payment Team: @corepayment-team
   - Product Owner: @Luke Lanterme
   - Engineering Manager: @Phuoc (Alex) Nguyen

Emergency Escalation

For Production Issues Affecting Customers:

Immediate Actions (< 5 minutes):

1. Post in #tx-core-payment-hub with "🚨 PRODUCTION ISSUE" tag
2. Include error details and customer impact
3. Tag @corepayment-team immediately

If No Response (< 30 minutes):

1. Escalate through your team lead
2. Tag Product Owner @Luke Lanterme
3. Consider paging on-call if critical

If Still No Response (< 1 hour):

1. Contact Engineering Manager @Phuoc (Alex) Nguyen

2. Initiate formal incident response process
3. Document business impact

Documentation Resources

Official Integration Documents

1. **How to Integrate with MultiX - Core Payment Platform**
   - Primary integration guide
   - Step-by-step instructions
   - Common questions and answers
2. **Payment Authorization API Contract**
   - Complete OpenAPI 3.1.0 specification
   - Request/response schemas
   - Error codes and messages
3. **Integrating Payment Flows with Core Payment Systems**
   - Communication patterns explained
   - Sequence diagrams for all integration types
   - Payload examples and templates
4. **Provisioning Mambu Resources Guideline**
   - Banking system setup requirements
   - Resource provisioning instructions
   - CloudFormation templates
5. **Provisioning Payment Rules/PUC Guideline**
   - Payment Use Case configuration
   - Business rules setup
   - Limit configuration

Code Repositories & Tools

1. **Payment Resource Blueprint**
   - Infrastructure as Code templates
   - PUC configuration examples
   - Deployment scripts
2. **AWS Signature V4 Library (Java)**
   - Authentication implementation
   - Request signing examples

- Integration with Spring Boot

3. **Example Integration Repository**
   - Real implementation examples
   - Configuration samples
   - Best practices demonstrated

Event Contracts & Schemas

1. **Event Contract - Payment Order Event**
   - Async payment result events
   - Event schema definitions
   - Consumer implementation examples
2. **Event Contract - Payment Auth Account Event**
   - Account balance change events
   - Real-time balance updates
   - Account lifecycle events
3. **MultiX - API Contract - Current Account**
   - Account management APIs
   - Account creation and updates
   - Status management operations

Team Contacts & Ownership

Core Payment Team (Cypher + Thunder)

- **Engineering Contact**: @Hieu Pham
- **Product Contact**: @Luke Lanterme
- **Team Responsibility**: Core payment processing, authorization, and platform management

Related Team Contacts

Service	Team	Engineering Contact	Product Contact
Notification Engine	Loki	@Nam Le	@Riskho Wardiat
Document Engine	Dragon	@Loc Huynh	@An Phan
Activity History	Neo	@Son Nguyen [Neo]	@Luke Lanterme
Profile Management	Alpha	@Kiet To @Long Tran [EM]	@Anthony Blackie
Step-up Authentication	Bhutan/Hawkeye	@Trang Le	@Riskho Wardiat
Dashboard	Zoro	@Son Pham	@Riskho Wardiat

Monitoring & Observability

DataDog Dashboards

GoTyme (Philippines) - Production:

- **Payment Hub Service**: DataDog Dashboard
- **Core Payment Service**: DataDog Dashboard
- **API Gateway Metrics**: Monitor rate limiting and response times

TymeBank (South Africa) - Production:

- **Payment Hub Service**: DataDog Dashboard
- **Core Payment Service**: DataDog Dashboard

Key Metrics to Monitor

Application Health:

- Request success rate (target: >99.5%)
- Response time P95 (target: <5 seconds for sync, <1 second for async ack)
- Error rate by error code
- Rate limiting violations

Business Metrics:

- Payment volume and trends
- Transaction amounts and patterns
- Success rate by payment type
- Customer impact metrics

Infrastructure Metrics:

- API gateway performance
- Kafka event processing lag
- Database connection health
- AWS service availability

Alerting Best Practices

Critical Alerts (Immediate Response):

```json
{
  "alert": "Payment Success Rate Below 95%",
  "condition": "success_rate < 95% for 5 minutes",
  "severity": "CRITICAL",
  "notification": ["slack", "pagerduty"]
}
```

Warning Alerts (Monitor Closely):

```json
{
  "alert": "Payment Response Time High",
  "condition": "p95_response_time > 10 seconds for 10 minutes",
  "severity": "WARNING",
  "notification": ["slack"]
}
```

Business Alerts (Track Trends):

```json
{
  "alert": "Payment Volume Anomaly",
  "condition": "hourly_volume deviates >50% from baseline",
  "severity": "INFO",
  "notification": ["slack"]
}
```

📋 Pre-Production Checklist

Integration Readiness Checklist

✅ Phase 1: Planning & Design

- ☐ **Business flow defined** with Business Analyst
- ☐ **Accounting flow documented** (which accounts debited/credited)
- ☐ **Payment Use Cases (PUCs) designed** and reviewed
- ☐ **Integration pattern selected** (Sync/Async + SMS/DMS)
- ☐ **Error handling strategy** defined
- ☐ **Rate limiting requirements** estimated

✅ Phase 2: Infrastructure Setup

- [ ] **Mambu resources provisioned** (GL accounts, transaction channels)
- [ ] **Payment Use Cases deployed** in all environments
- [ ] **API key obtained** with appropriate rate limits
- [ ] **Environment access validated** (DEV, TST, STG)
- [ ] **Security implementation completed** (AWS Signature V4)
- [ ] **Network connectivity verified** to all endpoints

✅ Phase 3: Development

- [ ] **Authentication implemented** with proper header handling
- [ ] **Idempotency mechanism built** with unique key generation
- [ ] **Request/response handling** implemented for all scenarios
- [ ] **Error handling** for all documented error codes
- [ ] **Retry logic** implemented with exponential backoff
- [ ] **Event consumers** implemented (for async patterns)
- [ ] **Logging and monitoring** instrumentation added

✅ Phase 4: Testing

- [ ] **Happy path testing** completed successfully
- [ ] **Error scenario testing** (insufficient funds, invalid accounts)
- [ ] **Network failure testing** (timeouts, retries)
- [ ] **Rate limiting testing** verified
- [ ] **Idempotency testing** (duplicate key handling)
- [ ] **Load testing** completed for expected volume
- [ ] **End-to-end testing** in UAT environment
- [ ] **Event processing testing** (for async patterns)

✅ Phase 5: Production Readiness

- [ ] **Security review** completed and approved
- [ ] **Performance testing** results acceptable
- [ ] **Monitoring and alerting** configured
- [ ] **Runbook created** for support team
- [ ] **Documentation updated** with production details
- [ ] **Rollback plan** documented and tested
- [ ] **Support team training** completed
- [ ] **Go-live communication** plan ready

Production Deployment Checklist

✅ Pre-Deployment (T-1 Week)

- [ ] **Production API key** obtained and validated

- [ ] **Production PUCs** deployed and tested
- [ ] **Production Mambu resources** provisioned
- [ ] **DataDog dashboards** configured for monitoring
- [ ] **Alerting rules** configured and tested
- [ ] **Support team** briefed on new integration

✅ Deployment Day

- [ ] **Health checks** passing in production
- [ ] **Test transactions** executed successfully
- [ ] **Monitoring** actively watching key metrics
- [ ] **Support team** on standby for issues
- [ ] **Rollback procedure** ready if needed

✅ Post-Deployment (T+1 Week)

- [ ] **Success metrics** meeting targets
- [ ] **Error rates** within acceptable thresholds
- [ ] **Customer feedback** positive or issues addressed
- [ ] **Performance** meeting SLA requirements
- [ ] **Documentation** updated with lessons learned

🎓 Best Practices & Guidelines

Security Best Practices

Authentication & Authorization

```javascript
// ✅ CORRECT: Secure credential management
const apiKey = process.env.CORE_PAYMENT_API_KEY; // From environment
const credentials = new AWS.EnvironmentCredentials(); // From IAM role

// ❌ WRONG: Hardcoded credentials
const apiKey = "SadeFPS5j0mM06C2VKUkwS241cDYRMYGjti6i91q"; // Never do this
```

Request Signing

javascript

```javascript
// ✅ CORRECT: Proper AWS signature implementation
const signature = aws4.sign({
  method: 'POST',
  path: '/internal/payment/auth',
  host: 'payment-platform-ingress.prd.tymeph.net',
  body: JSON.stringify(requestBody),
  headers: {
    'Content-Type': 'application/json',
    'x-api-key': apiKey
  }
}, credentials);

// ❌ WRONG: Skip signature validation
// Never send requests without proper AWS signatures
```

Input Validation

javascript

```javascript
// ✅ CORRECT: Validate all inputs
function validatePaymentRequest(request) {
  if (!request.paymentUseCase) {
    throw new Error('Payment use case is required');
  }
  if (!request.transactions || request.transactions.length === 0) {
    throw new Error('At least one transaction is required');
  }
  for (const txn of request.transactions) {
    if (txn.amount <= 0) {
      throw new Error('Transaction amount must be positive');
    }
    if (!txn.accountNo || !/^[a-zA-Z0-9]{5,15}$/.test(txn.accountNo)) {
      throw new Error('Invalid account number format');
    }
  }
}
```

Performance Best Practices

Connection Management

javascript

```javascript
// ✅ CORRECT: Reuse HTTP connections
const https = require('https');
const agent = new https.Agent({
  keepAlive: true,
  keepAliveMsecs: 30000,
  maxSockets: 50
});

const fetch = require('node-fetch');
const response = await fetch(url, {
  agent: agent,
  timeout: 15000
});
```

Async Processing

javascript

```javascript
// ✅ CORRECT: Use async for high volume
async function processPayments(payments) {
  const results = await Promise.allSettled(
    payments.map(payment => processAsyncPayment(payment))
  );

  // Handle results with proper error handling
  return results.map((result, index) => ({
    payment: payments[index],
    success: result.status === 'fulfilled',
    result: result.value || result.reason
  }));
}

// ❌ WRONG: Sequential processing for high volume
for (const payment of payments) {
  await processPayment(payment); // Slow for large batches
}
```

Caching Strategy

```javascript
// ✅ CORRECT: Cache account information
const accountCache = new Map();

async function getAccountWithCache(accountNo) {
  if (accountCache.has(accountNo)) {
    const cached = accountCache.get(accountNo);
    if (Date.now() - cached.timestamp < 300000) { // 5 minute TTL
      return cached.data;
    }
  }

  const account = await getAccountDetails(accountNo);
  accountCache.set(accountNo, {
    data: account,
    timestamp: Date.now()
  });

  return account;
}
```

Error Handling Best Practices

Comprehensive Error Classification

```javascript
class PaymentError extends Error {
  constructor(errorCode, message, retryable = false, customerMessage = null) {
    super(message);
    this.errorCode = errorCode;
    this.retryable = retryable;
    this.customerMessage = customerMessage || message;
  }
}

function classifyError(errorResponse) {
  const errorCode = errorResponse.errors[0]?.errorCode;

  switch (errorCode) {
    case '4001001': // Insufficient funds
      return new PaymentError(
        errorCode,
        'Insufficient account balance',
        false, // Not retryable
        'Please ensure sufficient balance and try again'
      );

    case '5000001': // Internal server error
      return new PaymentError(
        errorCode,
        'Internal server error',
        true, // Retryable
        'Temporary system issue, please try again'
      );

    default:
      return new PaymentError(
        errorCode || 'UNKNOWN',
        errorResponse.errors[0]?.errorMessage || 'Unknown error',
        false
      );
  }
}
```

Retry Logic Implementation

```javascript
async function retryPayment(paymentFn, maxRetries = 3) {
  for (let attempt = 1; attempt <= maxRetries; attempt++) {
    try {
      return await paymentFn();
    } catch (error) {
      const paymentError = classifyError(error);

      if (!paymentError.retryable || attempt === maxRetries) {
        throw paymentError;
      }

      // Exponential backoff with jitter
      const delay = Math.min(1000 * Math.pow(2, attempt - 1), 30000);
      const jitter = Math.random() * 1000;
      await new Promise(resolve => setTimeout(resolve, delay + jitter));

      console.log(`Payment attempt ${attempt} failed, retrying in ${delay + jitter}ms`
    }
  }
}
```

Monitoring & Observability Best Practices

Structured Logging

```javascript
const logger = require('winston');

function logPaymentAttempt(paymentRequest, orderId, status, duration) {
  logger.info('Payment processed', {
    orderId: orderId,
    paymentUseCase: paymentRequest.paymentUseCase,
    amount: paymentRequest.transactions.reduce((sum, txn) => sum + txn.amount, 0),
    currency: paymentRequest.transactions[0]?.currency,
    status: status,
    duration: duration,
    customerId: paymentRequest.tags?.CUSTOMER_ID,
    feature: paymentRequest.source,
    timestamp: new Date().toISOString()
  });
}
```

Custom Metrics

```javascript
const StatsD = require('node-statsd');
const metrics = new StatsD();

function recordPaymentMetrics(paymentRequest, result) {
  // Record payment volume
  metrics.increment('payments.total', 1, {
    payment_use_case: paymentRequest.paymentUseCase,
    status: result.status,
    currency: paymentRequest.transactions[0]?.currency
  });

  // Record payment amounts
  const totalAmount = paymentRequest.transactions.reduce((sum, txn) => sum + txn.amoun
  metrics.histogram('payments.amount', totalAmount, {
    payment_use_case: paymentRequest.paymentUseCase,
    currency: paymentRequest.transactions[0]?.currency
  });

  // Record response times
  metrics.timing('payments.response_time', result.duration);
}
```

Testing Best Practices

Unit Testing

javascript

```javascript
describe('Payment Service', () => {
  test('should generate unique idempotency keys', () => {
    const service = new PaymentService();
    const key1 = service.generateIdempotencyKey('CUST123', 'BillPay', 'REF001');
    const key2 = service.generateIdempotencyKey('CUST123', 'BillPay', 'REF002');

    expect(key1).not.toBe(key2);
    expect(key1).toContain('CUST123');
    expect(key1).toContain('BillPay');
  });

  test('should handle insufficient funds error', async () => {
    const mockResponse = {
      status: 400,
      errors: [{ errorCode: '4001001', errorMessage: 'Insufficient funds' }]
    };

    jest.spyOn(global, 'fetch').mockResolvedValue({
      ok: false,
      status: 400,
      json: () => Promise.resolve(mockResponse)
    });

    const service = new PaymentService();
    const result = await service.processPayment(mockPaymentRequest);

    expect(result.success).toBe(false);
    expect(result.errorCode).toBe('4001001');
  });
});
```

Integration Testing

```javascript
describe('Core Payment Integration', () => {
  test('should successfully process payment in test environment', async () => {
    const paymentRequest = {
      paymentUseCase: 'TEST_PaymentUseCase',
      paymentUseCaseVersion: '1.0.0',
      paymentDefinition: 'TEST_Payment',
      requestType: 'SMS',
      timestamp: Date.now().toString(),
      source: 'IntegrationTest',
      transactions: [
        {
          transactionDefinition: 'TEST_Debit',
          accountNo: process.env.TEST_ACCOUNT_NO,
          amount: 10.00,
          currency: 'PHP',
          narrative: 'Integration test payment'
        }
      ],
      journalEntries: [],
      tags: { TEST_FLAG: 'true' }
    };

    const service = new PaymentService({
      environment: 'TST',
      apiKey: process.env.TEST_API_KEY
    });

    const result = await service.processPayment(paymentRequest);

    expect(result.status).toBe('APPROVED');
    expect(result.orderId).toBeDefined();
    expect(result.transactions).toHaveLength(1);
  });
});
```

🏁 Conclusion

You now have a **complete understanding** of the Core Payment system and everything needed for successful integration. This guide has taken you through:

📚 What You've Learned

- **Foundation Knowledge**: Understanding what Core Payment is and why it exists
- **Architecture Mastery**: Deep knowledge of all system components and interactions

- **Integration Patterns**: All 4 communication patterns and when to use each
- **Setup & Configuration**: Complete provisioning and configuration process
- **API Implementation**: Step-by-step coding with real examples
- **Production Readiness**: Monitoring, troubleshooting, and best practices

🎯 Key Takeaways for Success

1. **Always Start with Business Analysis**
   - Never code before understanding the accounting flow
   - Work closely with your Business Analyst
   - Document all business rules and validations
2. **Follow the Integration Sequence**
   - Provision Mambu resources first
   - Create and test PUCs in lower environments
   - Implement security correctly from the start
   - Test thoroughly before production
3. **Implement Defensive Programming**
   - Validate all inputs and outputs
   - Handle all documented error scenarios
   - Implement proper retry logic
   - Use idempotency correctly
4. **Monitor Everything**
   - Set up comprehensive monitoring and alerting
   - Log structured data for debugging
   - Track business metrics and technical performance
   - Prepare for production support
5. **Leverage Support Resources**
   - Use Slack channels for real-time help
   - Create detailed support tickets when needed
   - Follow escalation procedures for production issues
   - Maintain good relationships with the Core Payment team

🚀 Your Next Steps

1. **Start Small**: Begin with a simple payment type (like airtime top-up)
2. **Test Thoroughly**: Use the provided checklists and testing guidelines

3. **Get Help Early**: Don't hesitate to reach out to support channels
4. **Scale Gradually**: Add complexity and volume incrementally
5. **Share Knowledge**: Document your learnings for your team

🌟 Success Metrics

Your integration is successful when you achieve:

- ✅ **>99.5% payment success rate** in production
- ✅ **<5 second response times** for synchronous calls
- ✅ **Zero security vulnerabilities** in security review
- ✅ **Complete error handling** for all scenarios
- ✅ **Comprehensive monitoring** and alerting
- ✅ **Happy customers** using your payment features

Remember: **Every payment integration is unique**, but following this guide ensures you cover all essential requirements for a secure, reliable, and scalable integration with Core Payment.

Good luck with your integration! 🎉

The Core Payment team and community are here to help you succeed. Don't hesitate to reach out whenever you need assistance.

📖 Complete Source Documentation

Primary Documentation Analyzed:

- Core Payment (Main Page) - *Empty placeholder*
- How to Integrate with MultiX - Core Payment Platform - *Primary integration hub*
- Integrating Payment Flows with Core Payment Systems - *Integration patterns*
- Payment Authorization API Contract - *Complete OpenAPI spec*
- Provisioning Mambu Resources Guideline - *Banking setup*
- Provisioning Payment Rules/PUC Guideline - *Payment use cases*
- Accessing Current Account Details Guideline - *Account APIs*
- Register API key for payment-platform-ingress - *API management*
- How to create a support ticket for Core Payment Platform - *Support process*
- Event Contract - Payment Auth Account Event - *Account events*
- Event Contract - Payment Order Event - *Payment events*
- MultiX - API Contract - Current Account - *Account management*

Additional Resources:

- Payment Resource Blueprint Repository
- AWS Signature V4 Implementation
- Example Integration Code

Analysis Timestamp: 2025-06-11 15:30:00 UTC