Billing Service - Payment Processing with Mercado Pago PIX

Overview

This microservice handles payment processing using **Hexagonal Architecture** (Ports and Adapters pattern). The service is responsible for:

- Listening to payment request messages from an AWS SQS queue
- Processing payments through Mercado Pago API using PIX payment method (sandbox environment)
- Persisting payment information with order and client IDs
- Publishing payment response messages to notify other services

Architecture

Queue-Based Payment Processing Flow

1. **Payment Request** → Another microservice publishes payment requests to AWS SQS queue
2. **Queue Listener** → Billing service listens and consumes messages from the queue
3. **Payment Processing** → Sends payment requests to Mercado Pago API (sandbox)
4. **Response Publishing** → Publishes payment status to response queue

Payment Request Message Format

```json
{
  "order_id": "uuid",
  "client_id": "uuid",
  "amount": 100.0,
  "customer_email": "customer@example.com",
  "customer_document": "12345678900",
  "customer_name": "Customer Name",
  "description": "Payment for order XYZ"
}
```

PROF

Hexagonal Architecture Layers

```
billing-service/
├── domain/                          # Core Business Logic (Center of
Hexagon)
│   ├── entity/                      # Domain Entities
│   │   └── Payment.java            # Payment entity with orderId,
clientId, amount, QR codes
│   ├── valueobject/                 # Value Objects
│   │   └── PaymentStatus.java     # PENDING, PROCESSING, APPROVED,
REJECTED, FAILED
│   ├── dto/                        # Domain DTOs
│   │   └── PaymentResponse.java   # Payment gateway response DTO
│   └── exception/                  # Domain Exceptions
│       └── PaymentProcessingException.java
│
├── application/                    # Application Use Cases
│   ├── port/
│   │   ├── in/                     # Input Ports (Use Case Interfaces)
│   │   │   └── ProcessPaymentUseCase.java
│   │   └── out/                    # Output Ports (Repository/External
Service Interfaces)
│   │       ├── PaymentGatewayPort.java
│   │       ├── PaymentRepositoryPort.java
│   │       └── PaymentResponseMessagePort.java
│   └── service/                    # Use Case Implementations
│       └── ProcessPaymentService.java
│
└── infrastructure/                 # Adapters (Outside of Hexagon)
    ├── adapter/
    │   ├── in/                     # Input Adapters
    │   │   ├── messaging/          # Queue Listener
    │   │   │   ├── PaymentQueueListener.java
    │   │   │   └── dto/
    │   │   │       └── PaymentRequestDto.java
    │   │   └── web/
    │   │       └── controller/
    │   │           ├── HealthController.java
    │   │           └── GlobalExceptionHandler.java
    │   └── out/                    # Output Adapters
    │       ├── messaging/          # AWS SQS/SNS Integration
    │       │   ├── PaymentResponseMessageAdapter.java
    │       │   └── SqsMessageSender.java
    │       ├── payment/            # Mercado Pago Integration
    │       │   └── MercadoPagoAdapter.java
    │       └── persistence/        # Database Integration
    │           ├── entity/
    │           │   └── PaymentEntity.java
    │           ├── repository/
    │           │   └── SpringDataPaymentRepository.java
    │           ├── mapper/
    │           │   └── PaymentMapper.java
    │           └── PaymentRepositoryAdapter.java
    └── config/                     # Configuration Classes
        ├── AwsConfig.java
        ├── DatabaseConfig.java
        ├── JacksonConfig.java
        └── WebConfig.java
```

PROF

Core Concepts

Hexagonal Architecture Benefits:

- **Technology Independence**: Business logic is isolated from infrastructure concerns
- **Testability**: Easy to test business logic without external dependencies
- **Flexibility**: Easy to swap implementations (e.g., change payment gateway)
- **Maintainability**: Clear separation of concerns

Ports and Adapters:

- **Ports (Interfaces)**: Define contracts for communication
  - **Input Ports**: Define use cases (e.g., ProcessPaymentUseCase)
  - **Output Ports**: Define external dependencies (e.g., PaymentGatewayPort)
- **Adapters**: Implement the ports
  - **Input Adapters**: Queue Listeners (PaymentQueueListener)
  - **Output Adapters**: Database Repositories, Payment Gateways (Mercado Pago), Message Publishers

Queue-Based Processing

Payment Request Queue

- **Queue Name**: `payment-request-queue`
- **Description**: Listens for payment requests from other microservices
- **Visibility Timeout**: 5 minutes (300 seconds)
- **Acknowledgment Mode**: ON_SUCCESS (message removed only after successful processing)
- **Message Format**:

```json
{
  "order_id": "uuid",
  "client_id": "uuid",
  "amount": 1500.0,
  "customer_email": "customer@example.com",
  "customer_document": "12345678900",
  "customer_name": "Customer Name",
  "description": "Payment for order XYZ"
}
```

PROF

Dead Letter Queue (DLQ)

- **Queue Name**: `payment-request-dlq`
- **Description**: Stores messages that failed processing after multiple attempts
- **Max Receive Count**: 3 (messages move to DLQ after 3 failed attempts)
- **Purpose**: Prevents infinite retry loops for permanently failing messages

DLQ Setup for LocalStack (Development):

```
```

```
# Run the setup script
chmod +x scripts/setup-sqs-dlq.sh
./scripts/setup-sqs-dlq.sh
```

This script creates:

- Main queue (`payment-request-queue`) with 5-minute visibility timeout
- Dead letter queue (`payment-request-dlq`)
- Redrive policy: maxReceiveCount=3

Message Idempotency

The service implements **defense-in-depth idempotency** to prevent duplicate payment processing:

1. **Application-Level Check**: Service checks for existing payment by `workOrderId` before creating new one
2. **Database Constraint**: Unique constraint on `work_order_id` column prevents duplicate insertions
3. **Duplicate Handling**: If duplicate detected, returns existing payment (safe retry)

Why Idempotency Matters:

- SQS can deliver messages more than once (at-least-once delivery)
- Visibility timeout expiry causes message redelivery
- Network issues may cause message reprocessing
- Multiple service instances may receive same message

What Happens on Duplicate:

- If payment already exists with status ≠ PENDING: Returns existing payment immediately
- If payment is PENDING: Continues processing (legitimate retry after crash)
- If race condition occurs: Database constraint prevents duplicate, existing payment returned

PROF

Logging:

- Duplicate detection events logged with WARN level
- Race condition handling logged with INFO level
- Includes workOrderId and payment status for troubleshooting

Payment Response Queue

- **Queue Name**: `payment-response-queue`
- **Description**: Publishes payment processing results to notify other services
- **Message Format**:

```
{
  "id": "uuid",
  "orderId": "uuid",
  "clientId": "uuid",
  "amount": 1500.0,
  "status": "APPROVED",
  "externalPaymentId": "1234567890",
  "paymentMethod": "pix",
  "qrCode": "00020101021243650016COM.MERCADOLIBRE...",
  "qrCodeBase64": "iVBORw0KGgoAAAANSUhEUgAA...",
  "createdAt": "2024-01-01T10:00:00",
  "processedAt": "2024-01-01T10:00:10"
}
```

Payment Entity

The Payment entity stores all payment-related information:

- **id**: Unique payment identifier (UUID)
- **orderId**: Order identifier from the order service (UUID)
- **clientId**: Client identifier (UUID)
- **amount**: Payment amount (BigDecimal)
- **status**: Payment status (PENDING, PROCESSING, APPROVED, REJECTED, FAILED)
- **externalPaymentId**: Mercado Pago payment ID
- **paymentMethod**: Payment method (pix)
- **qrCode**: PIX QR code string
- **qrCodeBase64**: PIX QR code as base64 image
- **createdAt**: Payment creation timestamp
- **processedAt**: Payment processing completion timestamp
- **errorMessage**: Error message if payment failed

Payment States

Payment Status Flow:

PROF

1. **PENDING**: Payment request received from queue
2. **PROCESSING**: Payment being processed by Mercado Pago API
3. **APPROVED**: Payment successfully approved
4. **REJECTED**: Payment rejected by Mercado Pago
5. **FAILED**: Technical error during processing

Integration with External Systems

AWS SQS

- **Payment Request Queue**: Receives payment requests from order service
- **Payment Response Queue**: Publishes payment processing results to notify other services

Mercado Pago API (Sandbox)

- **Environment**: Sandbox (for testing)

- **Payment Method**: PIX
- **Integration**: Via Mercado Pago SDK for Java (version 2.1.4)
- **Response Handling**:
  - Creates PIX payment with QR code
  - Returns payment status immediately
  - Generates QR code for customer to scan and pay
- **QR Code**: Returns both string format and base64 image for display

Technology Stack

- **Java 21**
- **Spring Boot 4.0.2**
- **Spring Data JPA**
- **PostgreSQL** (Production)
- **H2** (Development/Testing)
- **AWS SDK** (SQS, SNS)
- **Mercado Pago SDK**
- **Maven**

Running the Application

Option 1: Docker (Recommended)

The easiest way to run the application is using Docker:

```
# Quick start with interactive menu
./start.sh

# Or using Docker Compose directly
docker-compose up -d --build

# View logs
docker-compose logs -f

# Stop services
docker-compose down
```

PROF

What's included:

- Application running on port 8080
- PostgreSQL 17 database on port 5433
- Health checks and automatic restarts
- Volume persistence for database

Access Points:

- Application: http://localhost:8080
- Health Check: http://localhost:8080/actuator/health

- PostgreSQL: localhost:5433 (user: postgres, password: postgres123)

For more Docker commands and options, see DOCKER_GUIDE.md

Option 2: Using Makefile

```
# View all available commands
make help

# Start all services
make up

# Start database only
make db-only

# View logs
make logs

# Stop services
make down
```

Option 3: Local Development

If you prefer to run the application locally:

Prerequisites

- Java 21+
- Maven 3.8+
- PostgreSQL (for production)
- AWS credentials (for SQS/SNS integration)
- Mercado Pago credentials

PROF

Configuration

Create a `.env` file in the root directory:

```
# Database
DB_HOST=localhost
DB_PORT=5432
DB_NAME=billing_db
DB_USERNAME=postgres
DB_PASSWORD=postgres

# AWS
AWS_REGION=us-east-2
AWS_ACCESS_KEY_ID=your-access-key
AWS_SECRET_ACCESS_KEY=your-secret-key
```

```
AWS_SQS_WORK_ORDER_QUEUE=work-order-queue
AWS_SNS_PAYMENT_RESPONSE_TOPIC=payment-response-topic
AWS_SNS_STATUS_UPDATE_TOPIC=status-update-topic

# Mercado Pago
MERCADO_PAGO_ACCESS_TOKEN=your-mercado-pago-token
MERCADO_PAGO_PUBLIC_KEY=your-public-key

# Application
SPRING_PROFILES_ACTIVE=development
SERVER_PORT=8081
```

Build and Run

```
# Build the project
mvn clean package

# Run the application
mvn spring-boot:run

# Or run the JAR
java -jar target/billing-service-0.0.1-SNAPSHOT.jar
```

Docker Support

```
# Build Docker image
docker build -t billing-service .

# Run with Docker
docker run -p 8081:8081 --env-file .env billing-service
```

PROF

Development

Running Tests

```
mvn test
```

Local Development with H2 Database

Set `SPRING_PROFILES_ACTIVE=development` in `.env` to use in-memory H2 database.

API Testing with curl

```bash
# Process payment
curl -X POST http://localhost:8081/api/payments/process \
  -H "Content-Type: application/json" \
  -d '{
    "workOrderId": "123e4567-e89b-12d3-a456-426614174000"
  }'
```

Monitoring and Logging

- All requests include correlation IDs for tracing
- Structured logging with context information
- Error handling with detailed exception messages
- Payment processing events are logged for audit purposes

Future Enhancements

- [ ] Implement actual Mercado Pago PIX API integration
- [ ] Add payment retry mechanism for failed transactions
- [ ] Implement payment refund functionality
- [ ] Add payment history and reporting endpoints
- [ ] Implement webhook handler for async payment notifications from Mercado Pago
- [ ] Add payment analytics and metrics

License

This project is proprietary software developed for FIAP.