partnr

Build Production-Ready Payment Gateway with Async Processing and Webhooks

Back

Mandatory Task

Domain

Backend Development

Skills

API Development, Communication, Database Management, Deep Learning, Docker Compose Configuration, EDA, Exponential Backoff Implementation, HMAC Signature Generation, Management, Product, Public Relations, SDK Development, Solution Architecture, System Reliability, Webhook Implementation

Difficulty

Advanced

Tools

Bull, Celery, Docker, Express.Js, FastAPI, Go, Java, Node.Js, PostgreSQL, Python, React, Redis, RQ, Spring Boot, Webpack

Industries

Fintech

Submission Received

Your submission has been received and is pending review. You can still update your submission till the effective deadline.

Submitted on 14 Jan 2026, 09:40 pm

Deadline: **21 Jan 2026, 04:59 pm**

Overview | Instructions | Resources | Submit

Description

Objective

Building on your payment gateway core, you will now transform it into a production-ready system by implementing **asynchronous job processing**, **webhook delivery with retry mechanisms**, **embeddable JavaScript SDK**, and **refund management**. This deliverable focuses on architectural patterns that enable scalability and reliability in real-world payment systems.

You'll learn to use **message queues** (Redis + Bull/Celery) for background job processing, implement **event-driven architecture** through webhooks with HMAC signature verification, build **cross-origin embeddable widgets**, handle **idempotent API operations**, and manage **complex retry logic** with exponential backoff. These are advanced patterns used by companies like Stripe, Razorpay, and PayPal. Completing this deliverable demonstrates your ability to build resilient, scalable systems that can handle real production workloads—a skill that distinguishes senior engineers from junior developers.

Core Requirements

- **Asynchronous payment processing** using Redis-based job queues with worker services processing payments in the background
- **Webhook system** that delivers payment events to merchant URLs with HMAC signature verification and automatic retry logic (5 attempts with exponential backoff)
- **Embeddable JavaScript SDK** that merchants can integrate on their websites to accept payments via modal/iframe without redirects
- **Refund API** with full and partial refund support, processed asynchronously through job workers
- **Idempotency keys** on payment creation to prevent duplicate charges on network retries
- **Enhanced dashboard** with webhook configuration, delivery logs, manual retry functionality, and integration documentation

Implementation Details

Updated Docker Compose

Add Redis and worker services to your existing `docker-compose.yml`:

```
version: '3.8'

services:
  postgres:
    # ... (keep existing configuration)

  redis:
    image: redis:7-alpine
```

```
    container_name: redis_gateway
    ports:
      - "6379:6379"
    healthcheck:
      test: ["CMD", "redis-cli", "ping"]
      interval: 10s
      timeout: 3s
      retries: 5

  api:
    # ... (keep existing configuration)
    environment:
      # ... (existing env vars)
      REDIS_URL: redis://redis:6379
    depends_on:
      postgres:
        condition: service_healthy
      redis:
        condition: service_healthy

  worker:
    build:
      context: ./backend
      dockerfile: Dockerfile.worker
    container_name: gateway_worker
    environment:
      DATABASE_URL: postgresql://gateway_user:gateway_pass@postgres:5432/payment_gateway
      REDIS_URL: redis://redis:6379
    depends_on:
      postgres:
        condition: service_healthy
      redis:
        condition: service_healthy
      api:
        condition: service_healthy

  dashboard:
    # ... (keep existing configuration)

  checkout:
    # ... (keep existing configuration)
```

Updated Database Schema

Add these additional tables and modifications to your existing database schema:

Refunds Table:

- `id`: Refund identifier (string, up to 64 characters), primary key, format: "rfnd_" + 16 alphanumeric characters
- `payment_id`: Reference to payments table (string, up to 64 characters), required, foreign key to payments(id)
- `merchant_id`: Reference to merchants table (UUID), required, foreign key to merchants(id)

- `amount`: Refund amount in smallest currency unit (integer), required
- `reason`: Refund reason/description (text), optional
- `status`: Refund status (string, up to 20 characters), defaults to 'pending' (values: 'pending', 'processed')
- `created_at`: Creation timestamp, auto-set to current time
- `processed_at`: Processing completion timestamp, optional, set when status changes to 'processed'

Webhook Logs Table:

- `id`: Unique identifier (UUID format), primary key, auto-generated
- `merchant_id`: Reference to merchants table (UUID), required, foreign key to merchants(id)
- `event`: Event type identifier (string, up to 50 characters), required (e.g., "payment.success", "refund.processed")
- `payload`: Event payload data (JSON object), required
- `status`: Delivery status (string, up to 20 characters), defaults to 'pending' (values: 'pending', 'success', 'failed')
- `attempts`: Number of delivery attempts made (integer), defaults to 0
- `last_attempt_at`: Timestamp of last delivery attempt, optional
- `next_retry_at`: Timestamp for next retry attempt, optional, used for scheduling retries
- `response_code`: HTTP response code from merchant's webhook endpoint (integer), optional
- `response_body`: Response body from merchant's webhook endpoint (text), optional
- `created_at`: Creation timestamp, auto-set to current time

Idempotency Keys Table:

- `key`: Idempotency key string (string, up to 255 characters), primary key, scoped with merchant_id
- `merchant_id`: Reference to merchants table (UUID), required, foreign key to merchants(id)
- `response`: Cached API response (JSON object), required, stores the complete response for the request
- `created_at`: Creation timestamp, auto-set to current time
- `expires_at`: Expiration timestamp, required, set to created_at + 24 hours

Merchants Table Modification:

- Add new column `webhook_secret`: Webhook secret for HMAC signature generation (string, up to 64 characters), optional
- Update test merchant record: Set `webhook_secret` to `'whsec_test_abc123'` for the merchant with email `'test@example.com'`

Note: The `webhook_secret` column is only added in Deliverable 2. For Deliverable 1, the merchants table does not need this column. When implementing Deliverable 2, add this column to your existing merchants table.

Required Indexes:

- Index on `refunds.payment_id` for efficient payment refund queries
- Index on `webhook_logs.merchant_id` for efficient merchant webhook queries
- Index on `webhook_logs.status` for efficient status-based queries
- Index on `webhook_logs.next_retry_at` where status is 'pending' for efficient retry scheduling queries

Job Queue System

Worker Service Structure:

```
backend/
├── Dockerfile.worker
└── src/
    └── main/java/com/gateway/
        ├── workers/
        │   ├── PaymentWorker.java
        │   ├── WebhookWorker.java
        │   └── RefundWorker.java
        └── jobs/
            ├── ProcessPaymentJob.java
            ├── DeliverWebhookJob.java
            └── ProcessRefundJob.java
```

Job Types to Implement:

1. Process Payment Job:

- Create a background job worker that processes payment requests asynchronously
- Job receives payment ID as input parameter
- Implementation steps:
  1. Fetch payment record from database using the payment ID
  2. Simulate payment processing delay: wait 5-10 seconds (random within this range)
     - **Test Mode**: When `TEST_MODE=true`, use `TEST_PROCESSING_DELAY` value instead of random range (default: 1000ms if not set)
  3. Determine payment outcome randomly based on success rate:
     - UPI payments: 90% success rate (90% chance of success, 10% chance of failure)
     - Card payments: 95% success rate (95% chance of success, 5% chance of failure)
     - **Test Mode**: When `TEST_MODE=true`, use `TEST_PAYMENT_SUCCESS` to determine outcome (overrides random logic, default: true if not set)
  4. Update payment status in database:
     - If successful: set status to 'success'

- If failed: set status to 'failed' and populate error_code and error_description fields

5. Enqueue a webhook delivery job for the appropriate event:
    - If successful: enqueue webhook for 'payment.success' event
    - If failed: enqueue webhook for 'payment.failed' event
    - Include payment data in webhook payload

Note: Test mode support is required for automated evaluation to ensure deterministic job processing.

2. Deliver Webhook Job:

- Create a background job worker that delivers webhook events to merchant endpoints
- Job receives merchant ID, event type, and payload data as input parameters
- Implementation steps:
    1. Fetch merchant details from database using merchant ID:
        - Retrieve webhook_url (must be configured, skip if NULL)
        - Retrieve webhook_secret for signature generation
    2. Generate HMAC-SHA256 signature:
        - Use merchant's webhook_secret as the key
        - Use the JSON string representation of the payload (no whitespace changes)
        - Generate hex-encoded signature
    3. Send HTTP POST request to merchant's webhook_url:
        - Header: `X-Webhook-Signature: <generated_signature>`
        - Header: `Content-Type: application/json`
        - Body: JSON payload
        - Timeout: 5 seconds
    4. Log webhook attempt in webhook_logs table:
        - Record attempt number (increment from previous attempts)
        - Record response code and response body (if available)
        - Record last_attempt_at timestamp
        - If successful (HTTP 200-299): set status to 'success'
        - If failed: set status to 'pending' and increment attempts
    5. If delivery failed and attempts < 5:
        - Calculate next_retry_at timestamp based on retry schedule (see Webhook Retry Logic below)
        - Keep status as 'pending' for future retry
    6. If delivery failed and attempts >= 5:
        - Set status to 'failed' permanently
        - Stop retrying

3. Process Refund Job:

- Create a background job worker that processes refund requests asynchronously

- Job receives refund ID as input parameter
- Implementation steps:
    1. Fetch refund record from database using the refund ID
    2. Verify payment is in refundable state:
        - Payment status must be 'success'
        - Verify total refunded amount (sum of all refunds for this payment) does not exceed payment amount
    3. Simulate refund processing delay: wait 3-5 seconds (random within this range)
    4. Update refund status in database:
        - Set status to 'processed'
        - Set processed_at timestamp to current time
    5. If refund amount equals payment amount (full refund):
        - Optionally update payment record to reflect full refund status
    6. Enqueue webhook delivery job for 'refund.processed' event:
        - Include refund data in webhook payload

Webhook Retry Logic:

```
Attempt 1: Immediate
Attempt 2: After 1 minute
Attempt 3: After 5 minutes
Attempt 4: After 30 minutes
Attempt 5: After 2 hours

After 5 failed attempts, mark webhook as permanently failed
```

Test Mode for Webhook Retries (Required): Your implementation must support shorter retry intervals for testing via environment variable:

- `WEBHOOK_RETRY_INTERVALS_TEST=true` - Uses test intervals instead of production intervals

When `WEBHOOK_RETRY_INTERVALS_TEST=true`, use these intervals:

- Attempt 1: 0 seconds (immediate)
- Attempt 2: 5 seconds
- Attempt 3: 10 seconds
- Attempt 4: 15 seconds
- Attempt 5: 20 seconds

This allows complete retry cycle testing in under 1 minute instead of hours. This test mode is required for automated evaluation.

Note: Production code should use the standard intervals (1 min, 5 min, 30 min, 2 hr) when `WEBHOOK_RETRY_INTERVALS_TEST` is not enabled.

Updated API Endpoints

Modified Create Payment Endpoint

```
POST /api/v1/payments

Headers:
X-Api-Key: key_test_abc123
X-Api-Secret: secret_test_xyz789
Idempotency-Key: unique_request_id_123 (optional)
Content-Type: application/json

Request Body:
{
  "order_id": "order_NXhj67fGH2jk9mPq",
  "method": "upi",
  "vpa": "user@paytm"
}

Response 201:
{
  "id": "pay_H8sK3jD9s2L1pQr",
  "order_id": "order_NXhj67fGH2jk9mPq",
  "amount": 50000,
  "currency": "INR",
  "method": "upi",
  "vpa": "user@paytm",
  "status": "pending",
  "created_at": "2024-01-15T10:31:00Z"
}
```

Updated implementation requirements (for Deliverable 2):

- Create an endpoint handler for `POST /api/v1/payments` that requires authentication
- Extract and validate API credentials from request headers (same as Deliverable 1)
- Handle optional `Idempotency-Key` header:
  - If idempotency key is provided:
    1. Check if a record exists in idempotency_keys table with:
       - key matching the provided idempotency key
       - merchant_id matching the authenticated merchant
    2. If found and not expired (expires_at > current time):
       - Return the cached response from the idempotency_keys record
       - Do not process the request again
    3. If found but expired:
       - Delete the expired record
       - Treat as a new request and continue processing
    4. If not found:
       - Continue with normal processing
- Validate payment details (same validation as Deliverable 1)
- Create payment record in database:
  - Set status to 'pending' (changed from 'processing' in Deliverable 1)

- Store all payment fields
- Set timestamps

- Enqueue ProcessPaymentJob with payment ID:
  - Add job to background job queue (Redis-based)
  - Do not wait for job completion
  - Payment processing happens asynchronously
- If idempotency key was provided:
  - Store the API response in idempotency_keys table
  - Scope the key to merchant_id + idempotency_key combination
  - Set expires_at to created_at + 24 hours
- Return response immediately:
  - HTTP status code 201
  - JSON body with payment details (status will be 'pending')
  - Do not wait for payment processing to complete

New: Capture Payment Endpoint

```
POST /api/v1/payments/{payment_id}/capture

Headers:
X-Api-Key: key_test_abc123
X-Api-Secret: secret_test_xyz789
Content-Type: application/json

Request Body:
{
  "amount": 50000
}

Response 200:
{
  "id": "pay_H8sK3jD9s2L1pQr",
  "order_id": "order_NXhj67fGH2jk9mPq",
  "amount": 50000,
  "currency": "INR",
  "method": "upi",
  "status": "success",
  "captured": true,
  "created_at": "2024-01-15T10:31:00Z",
  "updated_at": "2024-01-15T10:32:00Z"
}

Error Response 400:
{
  "error": {
    "code": "BAD_REQUEST_ERROR",
    "description": "Payment not in capturable state"
  }
}
```

Implementation notes: Update `captured` field to `true` in payments table.

partnr

New: Create Refund Endpoint

```
POST /api/v1/payments/{payment_id}/refunds

Headers:
X-Api-Key: key_test_abc123
X-Api-Secret: secret_test_xyz789
Content-Type: application/json

Request Body:
{
  "amount": 50000,
  "reason": "Customer requested refund"
}

Response 201:
{
  "id": "rfnd_K9pL2mN4oQ5r",
  "payment_id": "pay_H8sK3jD9s2L1pQr",
  "amount": 50000,
  "reason": "Customer requested refund",
  "status": "pending",
  "created_at": "2024-01-15T10:33:00Z"
}

Error Response 400:
{
  "error": {
    "code": "BAD_REQUEST_ERROR",
    "description": "Refund amount exceeds available amount"
  }
}
```

Implementation requirements:

- Create an endpoint handler for `POST /api/v1/payments/{payment_id}/refunds` that requires authentication
- Extract and validate API credentials from request headers (same as other endpoints)
- Extract payment ID from URL path parameter
- Validate credentials and fetch payment:
  - Look up payment by payment_id
  - Ensure payment.merchant_id matches the authenticated merchant
  - If payment not found or doesn't belong to merchant, return 404 or 400 error
- Verify payment is refundable:
  - Payment status must be 'success' (only successful payments can be refunded)
  - If payment status is not 'success', return 400 with error code "BAD_REQUEST_ERROR"
- Calculate total already refunded:
  - Query all refunds for this payment_id

- Sum the amount field of all refunds with status 'processed' or 'pending'
- This represents the total amount already refunded or in process

- Validate refund amount:
  - Request body must contain `amount` (integer, required) and `reason` (string, optional)
  - Verify: requested_amount <= (payment.amount - total_refunded_amount)
  - If validation fails, return 400 with error code "BAD_REQUEST_ERROR" and description "Refund amount exceeds available amount"
- Generate refund ID:
  - Format: "rfnd_" followed by exactly 16 alphanumeric characters
  - Must be unique (check for collisions and regenerate if needed)
- Create refund record in database:
  - Set status to 'pending'
  - Store payment_id, merchant_id, amount, reason
  - Set created_at timestamp
- Enqueue ProcessRefundJob:
  - Add job to background job queue with refund ID
  - Do not wait for job completion
  - Refund processing happens asynchronously
- Return response:
  - HTTP status code 201
  - JSON body containing refund details including id, payment_id, amount, reason, status, and created_at

New: Get Refund Endpoint

```
GET /api/v1/refunds/{refund_id}

Headers:
X-Api-Key: key_test_abc123
X-Api-Secret: secret_test_xyz789

Response 200:
{
  "id": "rfnd_K9pL2mN4oQ5r",
  "payment_id": "pay_H8sK3jD9s2L1pQr",
  "amount": 50000,
  "reason": "Customer requested refund",
  "status": "processed",
  "created_at": "2024-01-15T10:33:00Z",
  "processed_at": "2024-01-15T10:33:05Z"
}
```

New: List Webhook Logs Endpoint

```
GET /api/v1/webhooks?limit=10&offset=0
```

```
Headers:
X-Api-Key: key_test_abc123
X-Api-Secret: secret_test_xyz789

Response 200:
{
  "data": [
    {
      "id": "550e8400-e29b-41d4-a716-446655440001",
      "event": "payment.success",
      "status": "success",
      "attempts": 1,
      "created_at": "2024-01-15T10:31:10Z",
      "last_attempt_at": "2024-01-15T10:31:11Z",
      "response_code": 200
    }
  ],
  "total": 1,
  "limit": 10,
  "offset": 0
}
```

New: Retry Webhook Endpoint

```
POST /api/v1/webhooks/{webhook_id}/retry

Headers:
X-Api-Key: key_test_abc123
X-Api-Secret: secret_test_xyz789

Response 200:
{
  "id": "550e8400-e29b-41d4-a716-446655440001",
  "status": "pending",
  "message": "Webhook retry scheduled"
}
```

Implementation: Reset attempts to 0, set status to 'pending', enqueue DeliverWebhookJob.

New: Job Queue Status Endpoint (Required for Evaluation)

```
GET /api/v1/test/jobs/status

No authentication required (test endpoint)

Response 200:
{
  "pending": 5,
  "processing": 2,
  "completed": 100,
  "failed": 0,
```

```
  "worker_status": "running"
}
```

partnr

Implementation requirements:

- Create an endpoint handler for `GET /api/v1/test/jobs/status` that does not require authentication
- Query the job queue (Redis) to get statistics:
  - `pending`: Count of jobs waiting to be processed
  - `processing`: Count of jobs currently being processed
  - `completed`: Count of successfully completed jobs (optional, can be approximate)
  - `failed`: Count of failed jobs (optional, can be approximate)
  - `worker_status`: String indicating if worker is running ("running" or "stopped")
- Return HTTP status code 200

Note: This test endpoint is required for automated evaluation. It provides visibility into job queue state without requiring direct Redis access.

Webhook Specification

Events to emit:

- `payment.created` - When payment record is created
- `payment.pending` - When payment enters pending state
- `payment.success` - When payment succeeds
- `payment.failed` - When payment fails
- `refund.created` - When refund is initiated
- `refund.processed` - When refund completes

Webhook Payload Format:

```
{
  "event": "payment.success",
  "timestamp": 1705315870,
  "data": {
    "payment": {
      "id": "pay_H8sK3jD9s2L1pQr",
      "order_id": "order_NXhj67fGH2jk9mPq",
      "amount": 50000,
      "currency": "INR",
      "method": "upi",
      "vpa": "user@paytm",
      "status": "success",
      "created_at": "2024-01-15T10:31:00Z"
    }
  }
}
```

Signature Generation:

Implement a function to generate webhook signatures for secure event delivery

- Algorithm: Use HMAC-SHA256 (Hash-based Message Authentication Code with SHA-256)
- Input parameters:
  - `payload`: The webhook payload as a JSON string (must be the exact string sent in HTTP body, no whitespace changes)
  - `webhookSecret`: The merchant's webhook_secret from the database
- Process:
  1. Use the merchant's webhook_secret as the HMAC key
  2. Use the JSON string representation of the payload as the data to sign
  3. Important: The JSON string must match exactly what will be sent in the HTTP request body (no pretty-printing, no whitespace changes)
  4. Generate HMAC-SHA256 hash of the payload using the secret
  5. Encode the hash as a hexadecimal string (lowercase or uppercase, both acceptable)
- Return: The hex-encoded signature string
- Example: For payload `{"event":"payment.success","data":{...}}` and secret `whsec_test_abc123`, generate signature like `a1b2c3d4e5f6...` (64 hex characters for SHA-256)

Webhook HTTP Request:

```
POST https://merchant-website.com/webhook

Headers:
Content-Type: application/json
X-Webhook-Signature: <generated_signature>

Body:
<JSON payload>
```

Embeddable SDK

Create a JavaScript SDK that merchants can include on their websites:

File structure:

```
checkout-widget/
├── src/
│   ├── sdk/
│   │   ├── PaymentGateway.js
│   │   ├── modal.js
│   │   └── styles.css
│   └── iframe-content/
│       └── CheckoutForm.jsx
├── webpack.config.js
```

```
└── dist/
    └── checkout.js (bundled output)
```

partnr

SDK API:

```
// File: PaymentGateway.js
class PaymentGateway {
  constructor(options) {
    // options: {
    //   key: 'key_test_abc123',
    //   orderId: 'order_xyz',
    //   onSuccess: function(response) { },
    //   onFailure: function(error) { },
    //   onClose: function() { }
    // }
    //
    // Implementation:
    // 1. Validate required options
    // 2. Store configuration
    // 3. Prepare iframe modal
  }

  open() {
    // 1. Create modal overlay
    // 2. Create iframe with src pointing to checkout page
    // 3. Set iframe attributes with data-testid="payment-iframe"
    // 4. Append modal to document body
    // 5. Set up postMessage listener for iframe communication
    // 6. Show modal
  }

  close() {
    // 1. Remove modal from DOM
    // 2. Call onClose callback if provided
  }
}

// Expose globally
window.PaymentGateway = PaymentGateway;
```

Usage by merchants:

```
<script src="https://cdn.yourgateway.com/checkout.js"></script>
<button id="pay-button">Pay Now</button>

<script>
document.getElementById('pay-button').addEventListener('click', function() {
  const checkout = new PaymentGateway({
    key: 'key_test_abc123',
    orderId: 'order_xyz',
    onSuccess: function(response) {
      console.log('Payment successful:', response.paymentId);
    },
```

```
    onFailure: function(error) {
      console.log('Payment failed:', error);
    }
  });

  checkout.open();
});
</script>
```

SDK modal HTML structure (for automated testing):

```
<div id="payment-gateway-modal" data-testid="payment-modal">
  <div class="modal-overlay">
    <div class="modal-content">
      <iframe
        data-testid="payment-iframe"
        src="http://localhost:3001/checkout?order_id=xxx&embedded=true"
      ></iframe>
      <button
        data-testid="close-modal-button"
        class="close-button"
      >
        ×
      </button>
    </div>
  </div>
</div>
```

Cross-origin communication:

```
// In iframe (checkout page)
function sendMessageToParent(type, data) {
  window.parent.postMessage({
    type: type,  // 'payment_success', 'payment_failed', 'close_modal'
    data: data
  }, '*');
}

// In SDK (parent page)
window.addEventListener('message', function(event) {
  if (event.data.type === 'payment_success') {
    this.onSuccess(event.data.data);
    this.close();
  } else if (event.data.type === 'payment_failed') {
    this.onFailure(event.data.data);
  }
});
```

Enhanced Dashboard Features

New: Webhook Configuration Page (`/dashboard/webhooks`):

```
<div data-testid="webhook-config">
  <h2>Webhook Configuration</h2>

  <form data-testid="webhook-config-form">
    <div>
      <label>Webhook URL</label>
      <input
        data-testid="webhook-url-input"
        type="url"
        placeholder="https://yoursite.com/webhook"
      />
    </div>

    <div>
      <label>Webhook Secret</label>
      <span data-testid="webhook-secret">whsec_test_abc123</span>
      <button data-testid="regenerate-secret-button">
        Regenerate
      </button>
    </div>

    <button data-testid="save-webhook-button" type="submit">
      Save Configuration
    </button>

    <button data-testid="test-webhook-button" type="button">
      Send Test Webhook
    </button>
  </form>

  <h3>Webhook Logs</h3>
  <table data-testid="webhook-logs-table">
    <thead>
      <tr>
        <th>Event</th>
        <th>Status</th>
        <th>Attempts</th>
        <th>Last Attempt</th>
        <th>Response Code</th>
        <th>Actions</th>
      </tr>
    </thead>
    <tbody>
      <tr data-testid="webhook-log-item" data-webhook-id="log_123">
        <td data-testid="webhook-event">payment.success</td>
        <td data-testid="webhook-status">success</td>
        <td data-testid="webhook-attempts">1</td>
        <td data-testid="webhook-last-attempt">
          2024-01-15 10:31:11
        </td>
        <td data-testid="webhook-response-code">200</td>
        <td>
          <button
            data-testid="retry-webhook-button"
            data-webhook-id="log_123"
```

```
          >
            Retry
          </button>
        </td>
      </tr>
    </tbody>
  </table>
</div>
```

New: API Documentation Page (`/dashboard/docs`):

```
<div data-testid="api-docs">
  <h2>Integration Guide</h2>

  <section data-testid="section-create-order">
    <h3>1. Create Order</h3>
    <pre data-testid="code-snippet-create-order">
<code>
curl -X POST http://localhost:8000/api/v1/orders \
  -H "X-Api-Key: key_test_abc123" \
  -H "X-Api-Secret: secret_test_xyz789" \
  -H "Content-Type: application/json" \
  -d '{
    "amount": 50000,
    "currency": "INR",
    "receipt": "receipt_123"
  }'
</code>
    </pre>
  </section>

  <section data-testid="section-sdk-integration">
    <h3>2. SDK Integration</h3>
    <pre data-testid="code-snippet-sdk">
<code>
&lt;script src="http://localhost:3001/checkout.js"&gt;&lt;/script&gt;
&lt;script&gt;
const checkout = new PaymentGateway({
  key: 'key_test_abc123',
  orderId: 'order_xyz',
  onSuccess: (response) => {
    console.log('Payment ID:', response.paymentId);
  }
});
checkout.open();
&lt;/script&gt;
</code>
    </pre>
  </section>

  <section data-testid="section-webhook-verification">
    <h3>3. Verify Webhook Signature</h3>
    <pre data-testid="code-snippet-webhook">
<code>
```

```
const crypto = require('crypto');

function verifyWebhook(payload, signature, secret) {
  const expectedSignature = crypto
    .createHmac('sha256', secret)
    .update(JSON.stringify(payload))
    .digest('hex');

  return signature === expectedSignature;
}
</code>
    </pre>
  </section>
</div>
```

Testing Your Implementation

Create a simple merchant test app to verify webhooks work:

```
// test-merchant/webhook-receiver.js
const express = require('express');
const crypto = require('crypto');

const app = express();
app.use(express.json());

app.post('/webhook', (req, res) => {
  const signature = req.headers['x-webhook-signature'];
  const payload = JSON.stringify(req.body);

  // Verify signature
  const expectedSignature = crypto
    .createHmac('sha256', 'whsec_test_abc123')
    .update(payload)
    .digest('hex');

  if (signature !== expectedSignature) {
    console.log('❌ Invalid signature');
    return res.status(401).send('Invalid signature');
  }

  console.log('✅ Webhook verified:', req.body.event);
  console.log('Payment ID:', req.body.data.payment.id);

  res.status(200).send('OK');
});

app.listen(4000, () => {
  console.log('Test merchant webhook running on port 4000');
});
```

Run this and configure your webhook URL to `http://host.docker.internal:4000/webhook` (on Mac/Windows) or `http://172.17.0.1:4000/webhook` (on Linux).

Common Mistakes

1. **Job queue not processing**: Ensure worker service is running and connected to Redis. Check that jobs are being enqueued correctly and workers are consuming them. Many students forget to start the worker container.

2. **Webhook signatures don't match**: The signature must be generated from the exact JSON string sent in the request body. Don't pretty-print or modify the JSON. Use the same string for both signature generation and HTTP body.

3. **Idempotency keys not working**: Keys must be checked before any database operations. Store the complete response (not just payment ID) and return it identically on subsequent requests with the same key.

4. **Exponential backoff incorrect**: Retry delays must follow this exact schedule:

   - Attempt 1: Immediate (0 delay)
   - Attempt 2: After 1 minute (60 seconds)
   - Attempt 3: After 5 minutes (300 seconds)
   - Attempt 4: After 30 minutes (1800 seconds)
   - Attempt 5: After 2 hours (7200 seconds)

   Store the `next_retry_at` timestamp in the database based on these delays.

5. **SDK not loading in parent page**: Ensure your webpack config outputs a UMD bundle that exposes `PaymentGateway` globally. Check the bundle works by loading it in a plain HTML file.

6. **PostMessage origin restrictions**: While `'*'` origin is acceptable for this project, in production you should validate `event.origin`. Document this security consideration in your README.

7. **Refund amount validation**: Must check total refunded amount across all refunds for a payment, not just individual refund amounts. A payment of ₹500 should not allow two ₹400 refunds.

8. **Webhook retry scheduling**: Use database `next_retry_at` field to schedule retries, don't use in-memory timers. Workers may restart and lose scheduled retries.

9. **Payment status from Deliverable 1**: In Deliverable 1, payments are created with status `'processing'` and processed synchronously. For Deliverable 2, update your payment creation flow to return status `'pending'` instead of `'processing'`, since processing now happens asynchronously via job workers. The status flow in Deliverable 2 is: `pending` → (worker processes) → `success` / `failed`.

10. **Missing captured field**: Add `captured BOOLEAN DEFAULT false` to payments table. This field tracks whether a successful payment has been captured for settlement.

FAQs

Q: Should webhooks be delivered even if merchant's webhook URL is not set?
A: No. Only create webhook logs and attempt delivery if `webhook_url` is configured for the merchant. If NULL, skip webhook delivery but still process the payment normally.

Q: How do I test webhook retries without waiting hours?
A: For development, you can use shorter retry intervals (e.g., 10s, 30s, 1m, 2m, 5m). Just ensure the production configuration uses the specified intervals: 1min, 5min, 30min, 2hr.

Q: Can I use a different job queue instead of Bull/Celery?
A: Yes, as long as it's Redis-based. Options include BullMQ (Node.js), RQ (Python), Sidekiq (Ruby), or Asynq (Go). The key requirement is reliable background job processing with retry support.

Q: How should the SDK be served to merchants?
A: Build the SDK into a single `checkout.js` file using webpack/rollup. Serve it as a static file from your checkout service on port 3001. Merchants can include it via: `<script src="http://localhost:3001/checkout.js"></script>`

Q: What if a webhook fails after 5 attempts?
A: Mark the webhook log status as `failed` permanently and stop retrying. Merchants can manually retry from the dashboard if needed using the retry button, which resets attempts to 0.

Q: Should refunds be instant or take time?
A: Simulate a 3-5 second processing delay in the RefundWorker, then update status to `processed`. This mimics real-world refund processing time.

Q: How do I handle partial refunds correctly?
A: Track all refunds for a payment. Before creating a new refund, sum all existing refunds for that payment and ensure `requested_amount + sum(existing_refunds) <= payment.amount`. Store each refund as a separate record.

Q: What should the idempotency key expiry time be?
A: 24 hours from creation. After expiry, the same key can be reused. Store expiry as `created_at + 24 hours` and check if `current_time < expires_at` before returning cached response.

Q: How do I test the SDK locally without HTTPS?
A: For local development, HTTP is fine. The SDK should work on `http://localhost:3001`. For production, you'd need HTTPS, but that's not required for this project.

Q: Should the SDK work on mobile browsers?
A: Yes, the modal/iframe should be responsive and work on mobile viewports. Test on Chrome DevTools mobile emulation to ensure it's usable on smaller screens.

Q: What if merchant's webhook endpoint returns 500 error?
A: Treat it as a failed delivery attempt. Log the response code (500), increment attempts, and schedule a retry if attempts < 5. Merchants can see the error in webhook logs.

partnr

partnr