Hashpay
TECHNICAL PUBLICATIONS SUITE

• OFFICIAL PARTNER SPECIFICATION MANUAL · A4 RELEASE

Complete Developer Integration & API Reference Specification

Comprehensive software engineering manual: API authentication, server-to-server order APIs, HMAC-SHA256 webhooks, lifecycle statuses, settlement, error codes, and production launch checklist.

DOCUMENT CONTROL MATRIX

DOCUMENT REFERENCE HP-DOC-2026-INT-V2.4	CLASSIFICATION Public Technical Specification
VERSION & STATUS v2.4 · Production Release	RELEASE DATE & LANGUAGE Oktober 2026 · EN

© 2026 Hashpay Inc. All Rights Reserved · docs.hashpay.id
Zero Floating-Point Financial Standard

DOCUMENT STRUCTURE

Table of Contents

GETTING STARTED

1.0 Overview Chapter 1

2.0 Authentication Chapter 2

IMPLEMENTATION

3.0 Backend integration Chapter 3

4.0 Frontend integration Chapter 4

5.0 Webhooks Chapter 5

LIFECYCLE & SETTLEMENT

6.0 Transaction lifecycle Chapter 6

7.0 Settlement & payout Chapter 7

8.0 Errors & status codes Chapter 8

TESTING & LAUNCH

9.0 Sandbox & faucet Chapter 9

10.0 Go live (production) Chapter 10

CHAPTER 1.0 · GETTING STARTED

Integration Guide

> *A comprehensive developer guide for integrating Hashpay crypto payment gateway: server-side order creation, hosted checkout and direct API flows, mobile webview deep linking, webhook handling, and production launch.*

Choose your integration path

Hashpay offers two primary integration models tailored to your business architecture and user experience requirements:

INTEGRATION PATH	BEST FOR	IMPLEMENTATION EFFORT
Hosted Payment Page (Webview)	Web e-commerce, mobile apps (Flutter, React Native, iOS, Android), and POS terminals. Hashpay manages token selection, wallet connections, live rate locking, and dynamic QR display.	1 server API call + redirect or webview modal
Direct API (Custom In-App UI)	Merchants requiring 100% white-label checkout screens inside their own native application or checkout flow. You design the UI while Hashpay provides rate locking and on-chain verification.	4 server API calls + custom front-end UI

Environment matrix & base URLs

Hashpay maintains strict isolation between Sandbox (Testnet) and Production (Mainnet). Ensure your backend client uses the appropriate base URL and API keys:

PARAMETER	SANDBOX (TESTNET)	PRODUCTION (MAINNET)
Base API URL	`https://api.hashpay.id/v2/testnet`	`https://api.hashpay.id/v2/mainnet`
Hosted Checkout URL	`https://webview.hashpay.id/sandbox`	`https://webview.hashpay.id`
Supported Blockchains	Ethereum (Sepolia), BNB Chain (Testnet), Tron (Shasta), Solana (Devnet)	Ethereum, BNB Chain, Tron, Solana
Settlement Assets	USDT & USDC (Testnet Faucet)	USDT & USDC (Real Mainnet Assets)
API Credentials	Testnet Secret Key (`sec_test_...`)	Mainnet Secret Key (`sec_live_...`)

End-to-end integration phases

1. **Order Creation & Rate Quotation** — Your server verifies order limits, creates an order via `POST /merchant/fiat/order`, and receives a hosted checkout session URL (`data.url`) and scoped key.

2. **Front-End Presentation** — Deliver the payment interface to the buyer: open the hosted page via browser redirect, iframe, or mobile in-app webview; or render custom QR codes via Direct API.
3. **Blockchain Monitoring & Rate Lock** — Hashpay locks the real-time crypto exchange rate for 15 minutes. Our real-time node watchers detect incoming blockchain transfers within 2 to 5 seconds.
4. **Webhook Notification & Fulfillment** — Once required block confirmations are reached, Hashpay sends a cryptographically signed HTTP POST webhook (`transaction.success`) to your server. Your server verifies the signature and fulfills the customer order.
5. **Reconciliation & Settlement** — Funds are automatically swept to your business settlement address or paid out directly to local Indonesian bank accounts.

> ⭐ **GOLDEN RULE OF PAYMENT INTEGRATION**
>
> Always treat the incoming webhook notification as the authoritative source of truth for payment status. Never release goods or mark invoices as paid based solely on the browser redirect or front-end return callback, as client-side signals can be tampered with or interrupted.

CHAPTER 2.0 · GETTING STARTED

Authentication

> *Hashpay has three authentication styles: an api-key header for transaction endpoints, merchant JWT for the dashboard, and an order-scoped key for hosted checkout.*

api-key header (transaction endpoints)

Server-to-server calls to order, rate, QR, and token endpoints use your **secret key** in the `api-key` header.

Header CODE

```
curl -X POST "https://api.hashpay.id/v2/mainnet/merchant/fiat/order" \
  -H "api-key: $HASHPAY_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{ "merchant_id": "10001", "order_no": "INV-2026-0001", "amount": "150000" }'
```

Merchant JWT (dashboard & management)

Profile, dashboard, disbursement, payment-link, and webhook-config endpoints use `Authorization: Bearer <access_token>` obtained from `POST /v2/auth/login`.

ENDPOINT	AUTH
`/orderLimits`, `/merchant/fiat/order`, `/checkOrderRate`, `/pgqr/fiat/create`, `/listToken`, `/merchant/settlement`	`api-key` header
`/merchant`, `/merchant/dashboard`, `/merchant/transaction`	Merchant JWT
`/merchant/disbursement`, `/merchant/payment-link`	Merchant JWT
`/rate`, `/faucet`, `/order/payment-link`	Public

Order-scoped key (hosted checkout)

The hosted payment page uses a short-lived, order-scoped key (`pl_…`, TTL 30 minutes) embedded in the checkout `url`. Never expose your real secret key in a client.

> 💡 **CATATAN PENTING**
>
> Endpoints like `getOrder` and `checkOrderRate` accept either your real api-key or a matching order-scoped key.

CHAPTER 3.0 · IMPLEMENTATION

Back-End Integration

> *Complete server-side integration guide: query order limits, generate payment orders, lock crypto rates, create on-chain QR codes, and query transaction reconciliation states.*

Authentication & credential mapping

Every server-to-server call against Hashpay transaction endpoints requires your Secret Key sent in the `api-key` header. Never expose your Secret Key in client-side code, mobile bundles, or public repositories:

PARAMETER	HTTP HEADER / FIELD	DESCRIPTION & SOURCE
Secret Key	`api-key: <YOUR_SECRET_KEY>`	Your confidential merchant secret key. Located in Dashboard → Settings → API Keys.
Merchant ID	`merchant_id: "10001"`	Your public merchant identifier. Located in Dashboard → Settings → API Keys.
Order Number	`order_no` / `transactionID`	Your system's unique alphanumeric reference (e.g., `INV-2026-0001`). Reused consistently across calls.
Billing Amount	`amount: "150000"`	The fiat charge in IDR as an integer numeric string (without decimal points).
Token Contract	`tokenAddress`	Smart contract / mint address of the stablecoin, retrieved via `GET /listToken`.
Blockchain Network	`network`	Target blockchain network: `ETHEREUM`, `BNB Chain`, `TRON`, or `SOLANA`.

Step 0 — Check dynamic order limits (Pre-flight)

`GET` `/orderLimits?merchant_id={merchant_id}`

Before presenting payment options or creating an order, query your store's configured order limits. Hashpay calculates minimum order thresholds dynamically to ensure transaction amounts cover MDR fees and network gas costs:

Check Order Limits Request — BASH

```
curl -X GET "https://api.hashpay.id/v2/mainnet/orderLimits?merchant_id=10001" \
  -H "api-key: sec_live_9f83b27a8e..."
```

Check Order Limits Response JSON


```
{
  "status_code": 200,
  "message": "success",
  "data": {
    "merchant_id": "10001",
    "currency": "IDR",
    "min_order_amount": "10000.00",
    "max_order_amount": "500000000.00",
    "mdr_fee_percent": "0.50"
  }
}
```


> 💡 **CATATAN PENTING**
>
> Always validate that the customer's cart total is greater than or equal to `min_order_amount`. Submitting orders below the threshold will result in an immediate `400 Bad Request` validation failure.

Step 1 — Create an order (Both paths)

`POST` `/merchant/fiat/order`

The primary entry point for both **Path A (Hosted Page)** and **Path B (Direct API)**. Creates an active order record in the Hashpay system and issues an order-scoped checkout credential:

Create Order (cURL) BASH

```
curl -X POST "https://api.hashpay.id/v2/mainnet/merchant/fiat/order" \
  -H "api-key: sec_live_9f83b27a8e..." \
  -H "Content-Type: application/json" \
  -d '{
    "merchant_id": "10001",
    "order_no": "INV-2026-0001",
    "amount": "150000",
    "name": "Budi Santoso",
    "email": "budi@example.com",
    "phone": "081234567890",
    "products": [
      { "name": "Specialty Coffee Beans 250g", "price": 150000, "quantity": 1 }
    ]
  }'
```

Create Order (Node.js / TypeScript) TYPESCRIPT

```typescript
import axios from "axios";

interface CreateOrderResponse {
  status_code: number;
  message: string;
  data: {
    request: {
      order_no: string;
      merchant_id: string;
      amount: string;
      currency: string;
      status: string;
    };
    url: string;
    key: string;
    expired_at: number;
  };
}

async function createHashpayOrder(orderNo: string, amountIdr: number, customerEmail: string) {
  const response = await axios.post<CreateOrderResponse>(
    "https://api.hashpay.id/v2/mainnet/merchant/fiat/order",
    {
      merchant_id: process.env.HASHPAY_MERCHANT_ID,
      order_no: orderNo,
      amount: amountIdr.toString(),
      email: customerEmail,
    },
    {
      headers: {
        "api-key": process.env.HASHPAY_SECRET_KEY,
        "Content-Type": "application/json",
      },
    }
  );

  return response.data.data;
}
```

Create Order (PHP / Laravel) PHP

```php
use Illuminate\Support\Facades\Http;

$response = Http::withHeaders([
    'api-key' => config('services.hashpay.secret_key'),
    'Content-Type' => 'application/json',
])->post('https://api.hashpay.id/v2/mainnet/merchant/fiat/order', [
    'merchant_id' => config('services.hashpay.merchant_id'),
    'order_no'    => 'INV-2026-0001',
    'amount'      => '150000',
    'name'        => 'Budi Santoso',
    'email'       => 'budi@example.com',
]);

if ($response->successful()) {
    $data = $response->json('data');
    // For Path A: redirect customer to $data['url']
    return redirect($data['url']);
}
```

Create Order Response Payload JSON


```json
{
  "status_code": 201,
  "message": "success",
  "data": {
    "request": {
      "order_no": "INV-2026-0001",
      "merchant_id": "10001",
      "amount": "150000",
      "currency": "IDR",
      "status": "Created"
    },
    "url": "https://webview.hashpay.id/sandbox?key=pl_eyJhbGciOi...",
    "key": "pl_eyJhbGciOi...",
    "expired_at": 1821387856
  }
}
```


⭐ **PATH A DEVELOPERS: YOU ARE DONE WITH SERVER SETUP!**

If you choose **Path A (Hosted Checkout)**, you do not need to call steps 2 through 4. Simply forward `data.url` to your frontend application or redirect the buyer's browser. Hashpay's hosted interface automatically handles token selection, wallet pairing, rate locking, and QR code rendering. Proceed directly to **Front-End Integration**.

Path B — Direct API (Build your own checkout UI)

If your application renders custom payment UI components instead of opening Hashpay's hosted webview, execute steps 2, 3, and 4 on your server:

Step 2 — Retrieve supported tokens & wallets

`GET` `/listToken?merchant_id={merchant_id}`

Returns all active stablecoins (USDT, USDC), blockchain networks (Ethereum, BNB Chain, Tron, Solana), contract addresses, decimals, and supported wallet options configured for your merchant account:

List Token Request BASH

```bash
curl -X GET "https://api.hashpay.id/v2/mainnet/listToken?merchant_id=10001" \
  -H "api-key: sec_live_9f83b27a8e..."
```

List Token Response (Excerpt) JSON


```json
{
  "status_code": 200,
  "message": "success",
  "data": [
    {
      "walletType": "METAMASK",
      "walletIcon": "https://assets.hashpay.id/wallets/metamask.png",
      "tokenDetails": [
        {
          "name": "USDT",
          "network": "BNB Chain",
          "tokenAddress": "0x55d398326f99059fF775485246999027B3197955",
          "icon": "https://assets.hashpay.id/tokens/usdt.png",
          "decimals": 18,
          "chainId": 56
        },
        {
          "name": "USDT",
          "network": "TRON",
          "tokenAddress": "TR7NHqjeKQxGTCi8q8ZY4pL8otSzgjLj6t",
          "icon": "https://assets.hashpay.id/tokens/usdt.png",
          "decimals": 6,
          "chainId": 728126428
        }
      ]
    }
  ]
}
```


Step 3 — Lock exchange rate & compute fees

`POST` `/merchant/fiat/checkOrderRate`

Locks the real-time cryptocurrency exchange rate for 15 minutes and calculates the exact crypto amount required. Use the `tokenAddress` and `network` obtained from Step 2, and supply your `order_no` as `transactionID`:

Check Order Rate Request BASH

```bash
curl -X POST "https://api.hashpay.id/v2/mainnet/merchant/fiat/checkOrderRate" \
  -H "api-key: sec_live_9f83b27a8e..." \
  -H "Content-Type: application/json" \
  -d '{
    "merchant_id": "10001",
    "transactionID": "INV-2026-0001",
    "tokenAddress": "0x55d398326f99059fF775485246999027B3197955",
    "network": "BNB Chain"
  }'
```

Check Order Rate Response JSON


```json
{
  "status_code": 200,
  "message": "success",
  "data": {
    "order_no": "INV-2026-0001",
    "merchant_id": "10001",
    "status": "Created",
    "currency": "IDR",
    "amount": "150000",
    "amount_crypto": "9.200000",
    "crypto_rate": "16304.34",
    "is_direct_deposit": false,
    "merchant_fees": [
      {
        "name": "MDR Platform Fee",
        "fee_type": "PERCENT",
        "fee_value": "0.50",
        "fee_amount": "750.00",
        "fee_amount_crypto": "0.046000",
        "is_included": true
      }
    ],
    "total_fee_amount_merchant": "750.00",
    "total_fee_amount_crypto_merchant": "0.046000",
    "merchant_received_amount": "149250.00",
    "merchant_received_amount_crypto": "9.154000",
    "fees": [],
    "total_fee_amount": "0.00",
    "total_fee_amount_crypto": "0.000000",
    "total_amount": "150000.00",
    "total_amount_crypto": "9.200000"
  }
}
```


💡 **UNDERSTANDING DIRECT DEPOSIT VS HD DERIVATION**

When `is_direct_deposit: true`, the payment uses a shared deposit pool where a 3-digit unique code is appended to the nominal (e.g., 9.200470 USDT) to route the transfer to your specific invoice. When `is_direct_deposit: false`, a dedicated per-order HD address is generated exclusively for this transaction.

Step 4 — Generate on-chain QR & destination address

`POST` `/pgqr/fiat/create`

Generates the dynamic, payment-ready QR code data. Returns an EIP-681 standard wallet URI (`qrValue`) and the exact destination blockchain deposit address:

Create Dynamic QR Request BASH

```
curl -X POST "https://api.hashpay.id/v2/mainnet/pgqr/fiat/create" \
  -H "api-key: sec_live_9f83b27a8e..." \
  -H "Content-Type: application/json" \
  -d '{
    "merchant_id": "10001",
    "transactionID": "INV-2026-0001",
    "tokenAddress": "0x55d398326f99059fF775485246999027B3197955",
    "network": "BNB Chain",
    "walletType": "METAMASK"
  }'
```

Create Dynamic QR Response JSON

```
{
  "status_code": 200,
  "message": "success",
  "data": {
    "qrValue": "ethereum:0x55d398326f99059fF775485246999027B3197955@56/transfer?
address=0x2AC2223419f32a6Aab57D9d4cb3210C97Bc62d51&uint256=9200000000000000000",
    "request": {
      "order_no": "INV-2026-0001",
      "merchantID": "10001",
      "tokenName": "USDT",
      "network": "BNB Chain",
      "walletType": "METAMASK",
      "destinationAddress": "0x2AC2223419f32a6Aab57D9d4cb3210C97Bc62d51",
      "amount": "9.200000",
      "totalAmount": "9.200000",
      "totalAmountFiat": "150000.00",
      "uniqueCode": "0",
      "expiredAt": 1821388756,
      "is_direct_deposit": false
    }
  }
}
```

> ⚠ **CUSTOMER TRANSFER EXACTNESS**
>
> The buyer must transfer the exact `request.totalAmount` to `request.destinationAddress`. Any partial transfer or underpayment will prevent automatic completion and flag the invoice as Underpaid.

Step 5 — Query order status & reconciliation (Fallback)

`POST` `/merchant/fiat/getOrder`

While webhooks are the primary delivery channel for payment events, use `getOrder` as an on-demand reconciliation fallback (for example, if your webhook endpoint experienced downtime or when verifying an invoice status during customer support inquiries):

Get Order Request BASH

```
curl -X POST "https://api.hashpay.id/v2/mainnet/merchant/fiat/getOrder" \
  -H "api-key: sec_live_9f83b27a8e..." \
  -H "Content-Type: application/json" \
  -d '{
    "merchant_id": "10001",
    "order_no": "INV-2026-0001"
  }'
```

Get Order Response JSON

```
{
  "status_code": 200,
  "message": "success",
  "data": {
    "order_no": "INV-2026-0001",
    "merchant_id": "10001",
    "merchant_name": "Toko Sejahtera Bersama",
    "currency": "IDR",
    "amount": "150000",
    "status": "Success",
    "paid_amount": "9.200000",
    "total_amount": "9.200000",
    "total_amount_fiat": "150000.00",
    "tx_hash": "0x3a7e28b14c995f088127394c8b66da982341209b0a1d48293c834017834bcdef",
    "link_explorer":
"https://bscscan.com/tx/0x3a7e28b14c995f088127394c8b66da982341209b0a1d48293c834017834bcdef",
    "expired_at": 1821388756,
    "createdAt": "2026-10-23T10: 15:00Z",
    "updatedAt": "2026-10-23T10: 16:42Z"
  }
}
```

CHAPTER 4.0 · IMPLEMENTATION

Front-End Integration

> *Presenting the checkout experience to your buyers: web browser redirect, responsive iframe modal, Midtrans-style JavaScript SDK (snap.js), and mobile app in-app webview deep linking.*

Overview of presentation options

When an order is created, your server receives `data.url` containing an encrypted, short-lived session token. Choose the presentation approach that best suits your user interface:

OPTION	INTERFACE PATTERN	RECOMMENDED APPLICATION
Option A	Browser Full Redirect	Traditional multi-page e-commerce websites and standard checkout carts.
Option B	Responsive Iframe Embed	Single-page web applications (SPAs) desiring an in-page checkout modal.
Option C	JavaScript SDK (`snap.js`)	Websites needing seamless modal popups with JavaScript lifecycle callbacks.
Option D	Mobile In-App WebView	iOS, Android, Flutter, and React Native mobile applications.

Option A — Full browser redirect (Recommended for web)

Redirect the user's browser directly to `data.url`. You can append an optional `&redirectUrl=` query parameter specifying the return destination on your website once payment concludes:

Client-side Web Redirect — JAVASCRIPT

```javascript
// Obtain data.url from your backend order creation endpoint
const checkoutUrl = data.url + "&redirectUrl=" + encodeURIComponent("https://mystore.com/orders/INV-2026-0001/complete");

// Redirect the customer
window.location.assign(checkoutUrl);
```

Option B — Responsive iframe embed

Embed the payment checkout directly inside your page layout or custom modal dialog. Ensure you include `allow="clipboard-write"` so users can copy deposit addresses with one click:

Iframe Modal Markup — HTML

```html
<iframe
  src="https://webview.hashpay.id/?key=pl_eyJhbGciOi...&redirectUrl=https%3A%2F%2Fmystore.com%2Fcomplete"
  style="width: 100%; height: 740px; border: 0; border-radius: 16px; overflow: hidden;"
  allow="clipboard-write"
></iframe>
```

Option C — JavaScript SDK (snap.js)

Load our lightweight Midtrans-Snap-style client library. The script exposes `window.pay` for production and `window.paytest` for sandbox testing:

Snap.js Integration HTML

```
<!-- Include the Hashpay Snap library in your HTML header -->
<script src="https://webview.hashpay.id/snap/snap.js"></script>

<script>
  function openHashpayModal(orderKey) {
    // For Sandbox, use window.paytest(); for Production, use window.pay()
    window.pay(
      { key: orderKey },
      {
        onSuccess: function (result) {
          console.log("Payment detected successfully:", result);
          window.location.href = "/payment-success?order=" + result.order_no;
        },
        onPending: function (result) {
          console.log("Transaction pending on-chain confirmation:", result);
        },
        onError: function (error) {
          console.error("Payment failed or aborted:", error);
          alert("Payment could not be completed. Please try again.");
        },
        onClose: function () {
          console.log("Customer closed the checkout modal without paying.");
        }
      }
    );
  }
</script>
```

Option D — Mobile in-app webview & deep link handling

When hosting the checkout webview inside native mobile applications (Flutter, React Native, Android Kotlin, or iOS Swift), Web3 wallet connection buttons trigger custom URI schemes such as `ethereum:`, `solana:`, `wc:`, `metamask:`, and `trust:`. Standard mobile webviews reject non-HTTP schemes by default. You MUST intercept these URLs and launch native wallet apps:

Flutter WebView Implementation (webview_flutter) DART

```dart
import 'package:flutter/material.dart';
import 'package:webview_flutter/webview_flutter.dart';
import 'package:url_launcher/url_launcher.dart';

class HashpayCheckoutScreen extends StatefulWidget {
  final String checkoutUrl;
  const HashpayCheckoutScreen({Key? key, required this.checkoutUrl}) : super(key: key);

  @override
  State<HashpayCheckoutScreen> createState() => _HashpayCheckoutScreenState();
}

class _HashpayCheckoutScreenState extends State<HashpayCheckoutScreen> {
  late final WebViewController _controller;

  @override
  void initState() {
    super.initState();
    _controller = WebViewController()
      ..setJavaScriptMode(JavaScriptMode.unrestricted)
      ..setNavigationDelegate(
        NavigationDelegate(
          onNavigationRequest: (NavigationRequest request) async {
            final uri = Uri.parse(request.url);
            // Intercept non-HTTP Web3 deep links (MetaMask, Trust, Phantom, etc.)
            if (!request.url.startsWith('http://') && !request.url.startsWith('https://')) {
              if (await canLaunchUrl(uri)) {
                await launchUrl(uri, mode: LaunchMode.externalApplication);
                return NavigationDecision.prevent;
              }
            }
            return NavigationDecision.navigate;
          },
        ),
      )
      ..loadRequest(Uri.parse(widget.checkoutUrl));
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text('Crypto Checkout')),
      body: WebViewWidget(controller: _controller),
    );
  }
}
```

React Native WebView Implementation (react-native-webview) TYPESCRIPT

```typescript
import React from 'react';
import { Linking } from 'react-native';
import { WebView } from 'react-native-webview';

export function HashpayCheckoutModal({ checkoutUrl, onPaymentClosed }) {
  return (
    <WebView
      source={{ uri: checkoutUrl }}
      javaScriptEnabled={true}
      domStorageEnabled={true}
      onShouldStartLoadWithRequest={(request) => {
        // Intercept Web3 wallet schemes
        if (!request.url.startsWith('http://') && !request.url.startsWith('https://')) {
          Linking.openURL(request.url).catch((err) =>
            console.warn('Could not launch crypto wallet app:', err)
          );
          return false; // Prevent internal webview navigation crash
        }
        return true;
      }}
    />
  );
}
```

Native Android Kotlin Implementation (WebViewClient) KOTLIN

```kotlin
webView.webViewClient = object : WebViewClient() {
    override fun shouldOverrideUrlLoading(view: WebView?, request: WebResourceRequest?): Boolean {
        val url = request?.url?.toString() ?: return false
        if (!url.startsWith("http://") && !url.startsWith("https://")) {
            try {
                val intent = Intent(Intent.ACTION_VIEW, Uri.parse(url))
                view?.context?.startActivity(intent)
                return true
            } catch (e: ActivityNotFoundException) {
                Toast.makeText(view?.context, "No compatible Web3 wallet installed",
Toast.LENGTH_SHORT).show()
                return true
            }
        }
        return false
    }
}
```

🔒 **ALWAYS ENFORCE SERVER-SIDE WEBHOOK VERIFICATION**

Front-end return handlers (such as `onSuccess` or return URL query strings) are purely for guiding user navigation. Do not dispatch goods or unlock user subscriptions until your server has received and verified the HMAC-SHA256 signature on the `transaction.success` webhook event.

CHAPTER 5.0 · IMPLEMENTATION

Webhooks

> *Hashpay delivers signed HTTP POST notifications to your webhook URL whenever a transaction transitions to a terminal state. Webhooks are the authoritative source of truth for payment reconciliation.*

Webhook event types

Hashpay dispatches real-time event payloads corresponding to critical lifecycle milestones of customer payments:

EVENT NAME	TRIGGER CONDITION	RECOMMENDED MERCHANT ACTION
`transaction.success`	Customer transfer is fully confirmed on-chain and credited to your merchant ledger.	Mark order as PAID; deliver digital goods or dispatch inventory.
`transaction.expired`	The 15-minute rate lock window elapsed without payment receipt.	Mark order as EXPIRED; prompt customer to create a fresh checkout quote.
`transaction.invalid`	An on-chain transfer was detected but failed validation (e.g., underpayment, wrong token, or unsupported network).	Log the transaction hash and flag the invoice for manual customer support review.
`webhook.test`	Synthetic simulation triggered manually via Dashboard (Settings → API Keys → Send Test Event).	Verify that your signature checking and HTTP 200 response handler work smoothly.

HTTP headers sent with every webhook

Every outbound webhook delivery includes standard cryptographic and routing headers:

HEADER	DESCRIPTION & FORMAT
`X-Hashpay-Event`	The event identifier (e.g., `transaction.success`).
`X-Hashpay-Timestamp`	Unix epoch timestamp in seconds when the webhook was generated.
`X-Hashpay-Signature`	HMAC-SHA256 signature in the format `sha256=<hex_digest>`.
`X-Hashpay-Signature-Version`	Signature version identifier (`v2`).
`X-Hashpay-Idempotency-Key`	Unique idempotency key (derived from transaction hash or delivery attempt) to safely deduplicate retried events.

JSON event payloads

Here are exact payload examples for each event type:

1. transaction.success Payload JSON

```
{
  "event": "transaction.success",
  "data": {
    "order_no": "INV-2026-0001",
    "merchant_id": "10001",
    "status": "Success",
    "token": "USDT",
    "network": "BNB Chain",
    "paid_amount": "9.200000",
    "final_amount": "9.154000",
    "tx_hash": "0x3a7e28b14c995f088127394c8b66da982341209b0a1d48293c834017834bcdef",
    "link_explorer":
"https://bscscan.com/tx/0x3a7e28b14c995f088127394c8b66da982341209b0a1d48293c834017834bcdef",
    "manual_review": false
  }
}
```

2. transaction.expired Payload JSON

```
{
  "event": "transaction.expired",
  "data": {
    "order_no": "INV-2026-0002",
    "merchant_id": "10001",
    "status": "Expired",
    "expired_at": 1821388756
  }
}
```

3. transaction.invalid Payload JSON

```
{
  "event": "transaction.invalid",
  "data": {
    "order_no": "INV-2026-0003",
    "merchant_id": "10001",
    "status": "Invalid",
    "token": "USDT",
    "network": "BNB Chain",
    "paid_amount": "5.000000",
    "tx_hash": "0x89ab12cd34ef567890123456789abcdef0123456789abcdef0123456789abcde",
    "link_explorer": "https://bscscan.com/tx/0x89ab...",
    "invalid_message": "Underpayment detected: expected 9.200000 USDT but received 5.000000 USDT",
    "manual_review": true
  }
}
```

4. webhook.test Payload JSON

```
{
  "event": "webhook.test",
  "test": true,
  "message": "This is a test event from the Hashpay dashboard.",
  "data": {
    "order_no": "INV-TEST-9999",
    "merchant_id": "10001",
    "status": "Success"
  }
}
```

Verifying the webhook signature (HMAC-SHA256)

The signature is computed over the string `<timestamp>.<raw_body>` using your merchant **Webhook Signing Secret** as the HMAC key. Validate the signature in your server before parsing the JSON content:

Node.js / Express Signature Verification TYPESCRIPT

```typescript
import crypto from "node:crypto";
import { Request, Response } from "express";

export function verifyHashpayWebhook(req: Request, res: Response) {
  const signature = req.headers["x-hashpay-signature"] as string;
  const timestamp = req.headers["x-hashpay-timestamp"] as string;
  const secret = process.env.HASHPAY_SIGNING_SECRET!;

  // 1. Replay attack defense: reject timestamps older than 5 minutes (300 seconds)
  const age = Math.abs(Math.floor(Date.now() / 1000) - Number(timestamp));
  if (!Number.isFinite(age) || age > 300) {
    return res.status(400).send("Expired or invalid timestamp");
  }

  // 2. Compute expected HMAC-SHA256 over timestamp + "." + raw body buffer
  // Note: req.rawBody must contain the raw unmodified request Buffer
  const expected = "sha256=" + crypto
    .createHmac("sha256", secret)
    .update(Buffer.concat([Buffer.from(timestamp + "."), req.body])) // or rawBody
    .digest("hex");

  // 3. Timing-safe comparison to prevent timing attacks
  const sigBuffer = Buffer.from(signature ?? "");
  const expBuffer = Buffer.from(expected);
  const isValid = sigBuffer.length === expBuffer.length && crypto.timingSafeEqual(sigBuffer, expBuffer);

  if (!isValid) {
    return res.status(401).send("Invalid webhook signature");
  }

  // 4. Handle event
  const payload = JSON.parse(req.body.toString());
  if (payload.event === "transaction.success") {
    console.log("Order paid successfully:", payload.data.order_no);
    // Fulfill order in database...
  }

  // 5. Acknowledge with 200 OK immediately
  return res.status(200).json({ received: true });
}
```

PHP / Laravel Signature Verification PHP

```php
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class HashpayWebhookController extends Controller
{
    public function handle(Request $request)
    {
        $signature = $request->header('X-Hashpay-Signature');
        $timestamp = $request->header('X-Hashpay-Timestamp');
        $secret    = config('services.hashpay.signing_secret');

        // 1. Reject timestamps older than 300 seconds
        if (abs(time() - (int)$timestamp) > 300) {
            return response()->json(['error' => 'Timestamp expired'], 400);
        }

        // 2. Compute expected HMAC
        $rawBody  = $request->getContent();
        $expected = 'sha256=' . hash_hmac('sha256', $timestamp . '.' . $rawBody, $secret);

        // 3. Timing-safe comparison
        if (!hash_equals($expected, (string)$signature)) {
            return response()->json(['error' => 'Invalid signature'], 401);
        }

        // 4. Process event
        $payload = $request->json()->all();
        if ($payload['event'] === 'transaction.success') {
            $orderNo = $payload['data']['order_no'];
            // Update order status in database...
        }

        return response()->json(['status' => 'success'], 200);
    }
}
```

Python (FastAPI / Flask) Signature Verification PYTHON

```python
import hmac
import hashlib
import time

def verify_hashpay_webhook(raw_body: bytes, timestamp_str: str, signature: str, secret: str) -> bool:
    try:
        timestamp = int(timestamp_str)
    except (ValueError, TypeError):
        return False

    # 1. Anti-replay check (5 minute window)
    if abs(time.time() - timestamp) > 300:
        return False

    # 2. Compute HMAC-SHA256
    signed_payload = str(timestamp).encode("utf-8") + b"." + raw_body
    expected_hex = hmac.new(
        secret.encode("utf-8"),
        signed_payload,
        hashlib.sha256
    ).hexdigest()
    expected_signature = f"sha256={expected_hex}"

    # 3. Constant-time comparison
    return hmac.compare_digest(expected_signature, signature or "")
```

Delivery retries & idempotency best practices

- **Immediate 200 OK Response** — Return an HTTP `200` or `204` status code within 3 seconds. Queue heavy tasks (such as inventory sync, email generation, or ERP integration) to a background worker.
- **Automatic Exponential Retries** — If your server returns an HTTP 5xx or times out, Hashpay automatically retries delivery up to 3 times with exponential backoff delays.
- **Idempotent Execution** — Network blips may cause duplicate delivery of the same event. Record the `data.order_no` and `X-Hashpay-Idempotency-Key` in your database. If you have already processed the order, simply return HTTP 200 without duplicating fulfillment.
- **Never Re-serialize Raw JSON** — Always compute HMAC over the raw, unparsed request bytes received from the network. Re-serializing parsed JSON can alter key order or whitespace and invalidate the signature.

CHAPTER 6.0 · LIFECYCLE & SETTLEMENT

Transaction Lifecycle & Statuses

> *Understand the discrete states that each payment order traverses from creation to final on-chain settlement, along with terminal event webhooks.*

Transaction state transitions

Every customer transaction moves through a defined state machine:

STATUS	CLASSIFICATION	MEANING & STATE DESCRIPTION
`Created`	Intermediate	Order created in database; checkout session active and awaiting customer payment.
`Pending`	Intermediate	Payment transfer has been detected on the blockchain; node watchers are awaiting required block confirmations.
`Success`	Terminal	Payment is fully confirmed on-chain. Merchant balance is credited; dispatches `transaction.success` webhook.
`Expired`	Terminal	The 15-minute rate quote window lapsed without payment; dispatches `transaction.expired` webhook.
`Failed`	Terminal	Blockchain transaction execution failed or reverted on-chain.
`Cancelled`	Terminal	Order was explicitly cancelled prior to payment completion.
`Invalid`	Terminal (Exception)	Payment was received with mismatched parameters (e.g. underpayment or wrong token); triggers ops manual review.

> 💡 **INTERMEDIATE VS TERMINAL STATES**
>
> Only terminal states (`Success` , `Expired` , `Invalid` , `Failed`) dispatch webhook notifications. Intermediate states (`Created` , `Pending`) can be monitored in real time via the hosted checkout interface or polled via `POST /merchant/fiat/getOrder` .

Exception handling & ops manual review

When a customer accidentally underpays or transfers an unsupported token, the order is flagged as `Invalid` with `manual_review: true` . Hashpay operations engineers review the exception queue:

- If the customer settles the remainder, the operator marks the order settled, dispatching a delayed `transaction.success` event with `manual_review: true` .
- If the customer requests a return, the funds are refunded directly to the sender address on-chain.

CHAPTER 7.0 · LIFECYCLE & SETTLEMENT

Settlement & Payouts

> *How funds received across per-order blockchain deposit addresses are aggregated, converted, and disbursed to your settlement wallet or Indonesian bank accounts.*

Automated deposit sweep architecture

When a customer pays an order, the transfer arrives at a unique on-chain deposit address. Hashpay's background sweep workers continuously monitor wallet balances:

1. **Immediate Ledger Credit** — As soon as the transfer confirms on-chain, your merchant available balance (`balance`) is credited immediately. You do not need to wait for gas sweeps before fulfilling customer orders.
2. **Automated Gas & Aggregation Sweep** — Background workers aggregate funds from individual deposit addresses into your central hot/cold treasury wallet based on configured economic gas thresholds.

Manual sweep trigger (api-key)

`POST` `/merchant/settlement`

Merchants can trigger an immediate on-demand sweep across all mature deposit addresses:

Trigger Manual Sweep — BASH

```bash
curl -X POST "https://api.hashpay.id/v2/mainnet/merchant/settlement" \
  -H "api-key: sec_live_9f83b27a8e..."
```

Outbound withdrawals (Disbursement to bank & crypto)

To withdraw accumulated revenue to your local Indonesian bank account (BCA, Mandiri, BRI, BNI) or external cold crypto wallet, use the disbursement endpoints authenticated with your Merchant JWT token:

- **Calculate Quote & Fee** — `GET /v2/mainnet/merchant/disbursement/calculate?amount=10000000&channel=BANK` returns precise exchange rates and bank gateway transfer fees.
- **Submit Withdrawal** — `POST /v2/mainnet/merchant/disbursement` creates the withdrawal instruction with 2FA authorization.

CHAPTER 8.0 · LIFECYCLE & SETTLEMENT

Errors & Status Codes

> *Standardized JSON error envelope, HTTP status definitions, and retry guidelines for Hashpay API endpoints.*

Unified JSON error envelope

All Hashpay API errors return a standard JSON envelope containing `status_code`, `message`, and `data: null`:

Error Envelope Example — JSON

```
{
  "status_code": 400,
  "message": "amount is below the minimum order limit of 10000 IDR",
  "data": null
}
```

HTTP status code reference

STATUS CODE	CATEGORY	DESCRIPTION & RECOMMENDED ACTION
`400`	Bad Request	Validation error (e.g. invalid amount, missing fields, or amount below limit). Check request payload.
`401`	Unauthorized	Invalid or missing `api-key` header, expired JWT token, or incorrect signature. Verify credentials in dashboard.
`403`	Forbidden	Action not allowed for your account tier or key permission level.
`404`	Not Found	Target resource, merchant ID, or order number does not exist.
`429`	Too Many Requests	Rate limit exceeded. Implement exponential backoff before retrying.
`500`	Internal Server Error	Transient server issue. Retry with exponential backoff; if persistent, contact Hashpay support.

⭐ **CATATAN PENTING**

When implementing automated retry loops for 429 or 5xx responses, always use exponential backoff with randomized jitter to prevent thundering herd problems on your integration server.

CHAPTER 9.0 · TESTING & LAUNCH

Sandbox & faucet

> *Test the full flow on testnet with free tokens before touching real funds. Formats are identical to production.*

Sandbox base URL

Use `https://api.hashpay.id/v2/testnet`. Sandbox orders, tokens, and webhooks are fully isolated from production.

Get test tokens

The faucet mints testnet tokens and gas to an address you control.

GET	/faucet
POST	/faucet
GET	/faucet/status?address=

> 💡 **CATATAN PENTING**
>
> Faucet is testnet-only and rate-limited. Use the returned tokens to pay a sandbox order end-to-end.

Simulate a payment

1. Create a testnet order and open the sandbox checkout (`/sandbox`).
2. Send the exact `totalAmount` from the faucet wallet to the deposit address.
3. Watch the order flip to `Success` and receive the `transaction.success` webhook.

> ⚠️ **CATATAN PENTING**
>
> Always send the exact `totalAmount` (including the 3-digit unique code). Underpayment is flagged `Underpaid`.

CHAPTER 10.0 · TESTING & LAUNCH

Go Live (Production Checklist)

> *Final preparation and launch verification checklist before transitioning your Hashpay integration to live mainnet transactions.*

Production configuration switch

1. **Switch Base API URL** — Point your backend API client from `https://api.hashpay.id/v2/testnet` to `https://api.hashpay.id/v2/mainnet`.
2. **Install Production Secret Keys** — Replace testnet API credentials with your live production `api-key` and `merchant_id` obtained from your verified merchant dashboard.
3. **Configure Production Webhook Endpoint** — Enter your public, secure HTTPS webhook endpoint (e.g., `https://mystore.com/api/hashpay-webhook`) in **Settings → API Keys & Webhooks**.
4. **Verify Checkout Hosted Domain** — Ensure your checkout redirect target resolves to `https://webview.hashpay.id/` (without the `/sandbox` route prefix).
5. **Bind Settlement Bank Account or Crypto Wallet** — Verify your bank details or cold wallet address in **Settings → Settlement** so your sales revenue sweeps correctly.

Pre-launch security checklist

- Secret Key is stored strictly in server environment variables / secret manager; never exposed in public client bundles or Git repositories.
- Incoming webhook signatures are cryptographically validated against `X-Hashpay-Signature` using your live Webhook Signing Secret.
- Webhook event handler is idempotent and rejects duplicate delivery using `X-Hashpay-Idempotency-Key` or `order_no` tracking.
- Webhook replay prevention is active: requests with `X-Hashpay-Timestamp` older than 300 seconds are rejected.
- Webhook endpoint returns HTTP `200` or `204` within 3 seconds, queuing background fulfillment tasks asynchronously.
- Reconciliation cron worker is active to poll `POST /merchant/fiat/getOrder` as an automated fallback for unhandled orders.

> 🔒 **LIVE CRYPTO TRANSACTIONS ARE IRREVERSIBLE**
>
> Blockchain transactions cannot be reversed or chargebacked. Ensure your internal order fulfillment ledger and webhook signature handlers are thoroughly tested in sandbox before accepting real customer payments.