ZenPays Documentation

Guides & Examples

SDK v0.6.0 · 23 documents
Edition 19 October 2026 · commit unknown
docs.zenpays.com

Contents

Getting Started
- Installation
- Quick Start
- Configuration
- Authentication

Guides
- Custom Checkout Integration
- Error Handling
- **Webhooks**
  - Webhooks Overview
  - Payment Webhooks
  - Payout Webhooks
  - Ramp Webhooks
  - KYC Webhooks
  - Other Webhooks
  - Signature Verification
- Testing
- TypeScript

Examples
- Basic Payment
- Checkout Flow
- On-Ramp (Buy) Flow
- Off-Ramp (Sell) Flow
- Payout Flow
- Refund Flow

Resources
- Postman Collection
- Using ZenPays docs with AI tools

PART

Getting Started

4 documents

GETTING STARTED

Installation

Get started with the ZenPays SDK in your project.

Requirements

JAVASCRIPT

Node.js 18.0 or later, plus npm, yarn, or pnpm

PYTHON

Python 3.8 or later, plus pip

Install the package

JAVASCRIPT

NPM

```
npm install @zenxdigitalholdings/zenpays
```

YARN

```
yarn add @zenxdigitalholdings/zenpays
```

PNPM

```
pnpm add @zenxdigitalholdings/zenpays
```

PYTHON

```
pip install zenpays
```

TypeScript Support

JAVASCRIPT

The SDK is written in TypeScript and includes complete type definitions out of the box. No additional `@types/*` package is needed.

```
import type { PaymentIntent } from '@zenxdigitalholdings/zenpays'
import { ZenPays } from '@zenxdigitalholdings/zenpays'

const zenpays = new ZenPays({
  apiKey: process.env.ZENPAYS_API_KEY!,
})

// Full type safety
const intent: PaymentIntent = await zenpays.payments.getPaymentIntent('pi_xxx')
```

PYTHON

The Python SDK includes full type hints and works seamlessly with mypy and other type checkers.

```python
from zenpays import ZenPays
from zenpays.types.payment import PaymentIntent
import os

zenpays = ZenPays(api_key=os.environ["ZENPAYS_API_KEY"])

# Full type hints support
intent: PaymentIntent = zenpays.payments.get_payment_intent("pi_xxx")
```

Browser Support

JAVASCRIPT

The SDK works in both Node.js and browser environments. For browser usage, you can import directly:

```html
<script type="module">
  import { ZenPays } from 'https://unpkg.com/@zenxdigitalholdings/zenpays@latest/dist/index.mjs'

  const zenpays = new ZenPays({
    apiKey: 'your-publishable-key',
  })
</script>
```

> **Warning**
> Never expose your secret API key in client-side code. Use publishable keys for browser environments.

PYTHON

The Python SDK is designed for server-side use and is not compatible with browser environments.

> **Tip**
> Use the JavaScript SDK for browser-based integrations and the Python SDK for server-side operations.

Next Steps

- Quick Start - Create your first payment
- Configuration - Learn about SDK options
- Authentication - Manage API keys

GETTING STARTED

Quick Start

Learn how to create your first payment with the ZenPays SDK.

Initialize the Client

JAVASCRIPT

```javascript
import { ZenPays } from '@zenxdigitalholdings/zenpays'

const zenpays = new ZenPays({
  apiKey: 'your-api-key',
})
```

PYTHON

```python
from zenpays import ZenPays

zenpays = ZenPays(api_key="your-api-key")
```

Create a Payment Intent

A payment intent represents a single payment flow. Create one to start accepting payments:

JAVASCRIPT

```javascript
const paymentIntent = await zenpays.payments.createPaymentIntent({
  amount: 100,
  currency: 'INR',
  paymentMethod: 'upi',
  customerEmail: 'john@example.com',
  customerFirstName: 'John',
  customerLastName: 'Doe',
  customerCountry: 'IN',
  description: 'Premium subscription',
})

console.log('Payment Intent ID:', paymentIntent.intentId)
console.log('Payment Page URL:', paymentIntent.paymentPageUrl)
```

PYTHON

```python
payment_intent = zenpays.payments.create_payment_intent({
    "amount": 100,
    "currency": "INR",
    "paymentMethod": "upi",
    "customerEmail": "john@example.com",
    "customerFirstName": "John",
    "customerLastName": "Doe",
    "customerCountry": "IN",
    "description": "Premium subscription",
})

print("Payment Intent ID:", payment_intent["intentId"])
print("Payment Page URL:", payment_intent["paymentPageUrl"])
```

Confirm the Payment

Once the customer provides payment details, confirm the payment:

JAVASCRIPT

```javascript
const result = await zenpays.payments.confirmPayment(paymentIntent.intentId, {
  customerDetails: {
    name: 'John Doe',
    email: 'john@example.com',
    address: { country: 'US' },
  },
  paymentMethodDetails: {
    type: 'upi',
    vpa: 'john@upi',
  },
})

if (result.status === 'succeeded') {
  console.log('Payment successful!')
}
else if (result.nextAction) {
  // Handle 3D Secure or other required actions
  console.log('Redirect to:', result.nextAction.redirectUrl)
}
```

PYTHON

```python
result = zenpays.payments.confirm_payment(
    payment_intent["intent_id"],
    {
        "customer_details": {
            "name": "John Doe",
            "email": "john@example.com",
            "address": {"country": "US"},
        },
        "payment_method_details": {
            "type": "upi",
            "vpa": "john@upi",
        },
    },
)

if result["status"] == "succeeded":
    print("Payment successful!")
elif "next_action" in result:
    # Handle 3D Secure or other required actions
    print("Redirect to:", result["next_action"]["redirect_url"])
```

Check Payment Status

JAVASCRIPT

```javascript
const intent = await zenpays.payments.getPaymentIntent(paymentIntent.intentId)

console.log('Status:', intent.status)
// 'pending' | 'processing' | 'succeeded' | 'failed' | 'cancelled'
```

PYTHON

```python
intent = zenpays.payments.get_payment_intent(payment_intent["intent_id"])

print("Status:", intent["status"])
# 'pending' | 'processing' | 'succeeded' | 'failed' | 'cancelled'
```

Handle Webhooks

Set up webhooks to receive real-time payment updates:

JAVASCRIPT

```javascript
// Register a webhook endpoint
await zenpays.merchants.createWebhook({
  url: 'https://your-site.com/webhooks/zenpays',
  events: [
    'payment.intent.succeeded',
    'payment.intent.failed',
    'refund.completed',
  ],
})
```

PYTHON

```python
# Register a webhook endpoint
zenpays.merchants.create_webhook({
    "url": "https://your-site.com/webhooks/zenpays",
    "events": [
        "payment.intent.succeeded",
        "payment.intent.failed",
        "refund.completed",
    ],
})
```

Error Handling

The SDK throws specific error types for different scenarios:

JAVASCRIPT

```javascript
import { PaymentError, ValidationError, ZenPaysError } from '@zenxdigitalholdings/zenpays'

try {
  await zenpays.payments.createPaymentIntent({
    amount: 1000,
    currency: 'USD',
  })
}
catch (error) {
  if (error instanceof ValidationError) {
    console.error('Validation failed:', error.message)
  }
  else if (error instanceof PaymentError) {
    console.error('Payment failed:', error.message)
  }
  else if (error instanceof ZenPaysError) {
    console.error('API error:', error.message, error.code)
  }
}
```

PYTHON

```python
from zenpays.errors import PaymentError, ValidationError, ZenPaysError

try:
    zenpays.payments.create_payment_intent({
        "amount": 1000,
        "currency": "USD",
    })
except ValidationError as e:
    print(f"Validation failed: {e.message}")
except PaymentError as e:
    print(f"Payment failed: {e.message}")
except ZenPaysError as e:
    print(f"API error: {e.message} [{e.code}]")
```

Complete Example

Here's a complete example bringing it all together:

JAVASCRIPT

```javascript
import { ZenPays, ZenPaysError } from '@zenxdigitalholdings/zenpays'

async function processPayment() {
  const zenpays = new ZenPays({
    apiKey: process.env.ZENPAYS_API_KEY!,
  })

  try {
    // 1. Create payment intent
    const intent = await zenpays.payments.createPaymentIntent({
      amount: 2999,
      currency: 'INR',
      paymentMethod: 'upi',
      customerEmail: 'jane@example.com',
      customerFirstName: 'Jane',
      customerLastName: 'Smith',
      customerCountry: 'IN',
      description: 'Pro Plan - Monthly',
    })

    // 2. Confirm with payment details
    const result = await zenpays.payments.confirmPayment(intent.intentId, {
      customerDetails: {
        name: 'Jane Smith',
        email: 'jane@example.com',
        address: { country: 'US' },
      },
      paymentMethodDetails: {
        type: 'upi',
        vpa: 'jane@upi',
      },
    })

    // 3. Handle result
    if (result.status === 'succeeded') {
      console.log('Payment completed successfully!')
      console.log('Transaction ID:', result.externalTransactionId)
    }
    else if (result.nextAction?.type === 'redirect') {
      console.log('3D Secure required, redirect to:', result.nextAction.redirectUrl)
    }
  }
  catch (error) {
    if (error instanceof ZenPaysError) {
      console.error(`Error [${error.code}]: ${error.message}`)
    }
    throw error
  }
}

processPayment()
```

PYTHON

```python
import os
from zenpays import ZenPays
from zenpays.errors import ZenPaysError


def process_payment():
    zenpays = ZenPays(api_key=os.environ["ZENPAYS_API_KEY"])

    try:
        # 1. Create payment intent
        intent = zenpays.payments.create_payment_intent({
            "amount": 2999,
            "currency": "INR",
            "paymentMethod": "upi",
            "customerEmail": "jane@example.com",
            "customerFirstName": "Jane",
            "customerLastName": "Smith",
            "customerCountry": "IN",
            "description": "Pro Plan - Monthly",
        })

        # 2. Confirm with payment details
        result = zenpays.payments.confirm_payment(
            intent["intent_id"],
            {
                "customer_details": {
                    "name": "Jane Smith",
                    "email": "jane@example.com",
                    "address": {"country": "US"},
                },
                "payment_method_details": {
                    "type": "upi",
                    "vpa": "jane@upi",
                },
            },
        )

        # 3. Handle result
        if result["status"] == "succeeded":
            print("Payment completed successfully!")
            print("Transaction ID:", result["external_transaction_id"])
        elif result.get("next_action", {}).get("type") == "redirect":
            print("3D Secure required, redirect to:", result["next_action"]["redirect_url"])

    except ZenPaysError as error:
        print(f"Error [{error.code}]: {error.message}")
        raise


if __name__ == "__main__":
    process_payment()
```

Next Steps

- Configuration - Customize SDK behavior
- Payments API (https://docs.zenpays.com/docs/api-reference/payments) - Explore all payment methods
- Error Handling - Handle errors gracefully

GETTING STARTED

Configuration

Learn how to configure the ZenPays SDK for your environment.

Configuration Options

JAVASCRIPT

```javascript
import { ZenPays } from '@zenxdigitalholdings/zenpays'

const zenpays = new ZenPays({
  // Required: Your API key
  apiKey: 'your-api-key',
  // here

  // Optional: API base URL (defaults to production)
  baseUrl: 'https://api.zenpays.com',

  // Optional: API version (defaults to 'v1')
  apiVersion: 'v1',

  // Optional: Request timeout in milliseconds (defaults to 30000)
  timeout: 30000,

  // Optional: Custom fetch implementation
  fetch: customFetch,
})
```

PYTHON

```python
from zenpays import ZenPays

zenpays = ZenPays(
    # Required: Your API key
    api_key="your-api-key",

    # Optional: API base URL (auto-detected from API key)
    base_url="https://api.zenpays.com",

    # Optional: Request timeout in seconds (defaults to 30)
    timeout=30,

    # Optional: HMAC secret salt for request signing
    secret_salt="your-secret-salt",
)
```

Environment Configuration

Production

JAVASCRIPT

```javascript
const zenpays = new ZenPays({
  apiKey: process.env.ZENPAYS_LIVE_API_KEY!,
})
```

PYTHON

```python
import os

zenpays = ZenPays(api_key=os.environ["ZENPAYS_LIVE_API_KEY"])
```

Sandbox/Testing

JAVASCRIPT

```javascript
const zenpays = new ZenPays({
  apiKey: process.env.ZENPAYS_TEST_API_KEY!,
  baseUrl: 'https://api.zenpays.com',
})
```

PYTHON

```python
import os

zenpays = ZenPays(
    api_key=os.environ["ZENPAYS_TEST_API_KEY"],
    base_url="https://api.zenpays.com",
)
```

Configuration Options Reference

JavaScript

Option	Type	Default	Description
`apiKey`	`string`	*Required*	Your ZenPays API key
`baseUrl`	`string`	`'https://api.zenpays.com'`	API base URL
`apiVersion`	`string`	`'v1'`	API version to use
`timeout`	`number`	`30000`	Request timeout in milliseconds
`fetch`	`typeof fetch`	`globalThis.fetch`	Custom fetch implementation

Python

Option	Type	Default	Description
`api_key`	`str`	*Required*	Your ZenPays API key
`base_url`	`str`	Auto-detected	API base URL
`timeout`	`int`	`30`	Request timeout in seconds
`secret_salt`	`str`	`None`	HMAC secret for request signing

Environment Auto-Detection:

- `zp_live_*` → `https://api.zenpays.com`
- `zp_test_*` → `https://api.zenpays.com` (sandbox; same host, routed by key)
- anything else → `http://localhost:3000`

Using Environment Variables

We recommend storing your API keys in environment variables:

JAVASCRIPT

```
# .env
ZENPAYS_API_KEY=zp_live_xxxxx
ZENPAYS_TEST_API_KEY=zp_test_xxxxx
```

```
// Use test key in development
const apiKey = process.env.NODE_ENV === 'production'
  ? process.env.ZENPAYS_API_KEY
  : process.env.ZENPAYS_TEST_API_KEY

const zenpays = new ZenPays({ apiKey: apiKey! })
```

PYTHON

```
# .env
ZENPAYS_API_KEY=zp_live_xxxxx
ZENPAYS_TEST_API_KEY=zp_test_xxxxx
```

```
import os

# Use test key in development
api_key = (
    os.environ["ZENPAYS_API_KEY"]
    if os.getenv("ENV") == "production"
    else os.environ["ZENPAYS_TEST_API_KEY"]
)

zenpays = ZenPays(api_key=api_key)
```

Timeout Configuration

Adjust timeouts for different scenarios:

JAVASCRIPT

```
// Longer timeout for batch operations
const zenpays = new ZenPays({
  apiKey: 'your-api-key',
  timeout: 120000, // 2 minutes
})
```

PYTHON

```python
# Longer timeout for batch operations
zenpays = ZenPays(
    api_key="your-api-key",
    timeout=120,  # 2 minutes (in seconds)
)
```

Multiple Instances

Create multiple client instances for different environments or merchants:

JAVASCRIPT

```javascript
// Production client
const prodClient = new ZenPays({
  apiKey: process.env.ZENPAYS_LIVE_KEY!,
})

// Sandbox client for testing
const testClient = new ZenPays({
  apiKey: process.env.ZENPAYS_TEST_KEY!,
  baseUrl: 'https://api.zenpays.com',
})

// Use the appropriate client based on context
const client = isTestMode ? testClient : prodClient
```

PYTHON

```python
import os

# Production client
prod_client = ZenPays(api_key=os.environ["ZENPAYS_LIVE_KEY"])

# Sandbox client for testing
test_client = ZenPays(
    api_key=os.environ["ZENPAYS_TEST_KEY"],
    base_url="https://api.zenpays.com",
)

# Use the appropriate client based on context
client = test_client if is_test_mode else prod_client
```

Context Manager Support

JAVASCRIPT

The JavaScript SDK doesn't require explicit cleanup, but you can implement similar patterns if needed.

PYTHON

The Python SDK supports context managers for automatic resource cleanup:

```python
with ZenPays(api_key="your-api-key") as zenpays:
    # Perform operations
    intent = zenpays.payments.create_payment_intent({
        "amount": 1000,
        "currency": "USD",
    })
    # HTTP session automatically closed after the block
```

Type Safety

JAVASCRIPT

The SDK is fully typed. Get IntelliSense and type checking:

```typescript
import type { ZenPaysConfig } from '@zenxdigitalholdings/zenpays'
import { ZenPays } from '@zenxdigitalholdings/zenpays'

// Type-safe configuration
const config: ZenPaysConfig = {
  apiKey: 'your-api-key',
  timeout: 30000,
}

const zenpays = new ZenPays(config)
```

PYTHON

The SDK includes comprehensive type hints for IDE support:

```python
from zenpays import ZenPays
from zenpays.types.payment import PaymentIntent

zenpays = ZenPays(api_key="your-api-key")

# Type checkers understand the return types
intent: PaymentIntent = zenpays.payments.get_payment_intent("pi_xxx")
```

HMAC Request Signing

JAVASCRIPT

HMAC signing is not currently supported in the JavaScript SDK.

PYTHON

For enhanced security, enable HMAC request signing:

```python
zenpays = ZenPays(
    api_key="your-api-key",
    secret_salt="your-secret-salt",  # Get this from your dashboard
)

# All requests will now include an X-Signature header
```

This adds an HMAC-SHA256 signature to each request, providing an additional layer of security.

Next Steps

- Authentication - Learn about API key management

- Error Handling - Handle errors gracefully
- Testing - Test your integration

GETTING STARTED

Authentication

Learn how to manage API keys and authenticate with the ZenPays API.

API Key Types

ZenPays uses two types of API keys:

Key Type	Prefix	Usage
Live	`zp_live_`	Production payments
Test	`zp_test_`	Sandbox testing

Getting Your API Key

1. Log in to your ZenPays Dashboard
2. Navigate to **Settings** → **API Keys**
3. Copy your API key

> **Warning**
> Keep your API keys secure. Never expose them in client-side code or commit them to version control.

Using the SDK

JAVASCRIPT

```
import { ZenPays } from '@zenxdigitalholdings/zenpays'

const zenpays = new ZenPays({
  apiKey: process.env.ZENPAYS_API_KEY!,
})
```

PYTHON

```
import os
from zenpays import ZenPays

zenpays = ZenPays(api_key=os.environ["ZENPAYS_API_KEY"])
```

Environment-Based Authentication

JAVASCRIPT

```javascript
// Determine environment from API key prefix
function getEnvironment(apiKey: string): 'live' | 'test' {
  return apiKey.startsWith('zp_live_') ? 'live' : 'test'
}

const zenpays = new ZenPays({
  apiKey: process.env.ZENPAYS_API_KEY!,
})

console.log('Environment:', getEnvironment(process.env.ZENPAYS_API_KEY!))
```

PYTHON

```python
import os

# Determine environment from API key prefix
def get_environment(api_key: str) -> str:
    if api_key.startswith("zp_live_"):
        return "live"
    elif api_key.startswith("zp_test_"):
        return "test"
    return "dev"

zenpays = ZenPays(api_key=os.environ["ZENPAYS_API_KEY"])

print(f"Environment: {get_environment(os.environ['ZENPAYS_API_KEY'])}")
```

IP Whitelisting

For enhanced security, whitelist IPs allowed to use your API keys:

JAVASCRIPT

```javascript
// List whitelisted IPs
const ips = await zenpays.merchants.listWhitelistedIPs()

// Add a new IP
await zenpays.merchants.addIPToWhitelist(
  '203.0.113.50',
  'Production Server'
)

// Remove an IP
await zenpays.merchants.removeIPFromWhitelist('ip_entry_id')
```

PYTHON

```python
# List whitelisted IPs
ips = zenpays.merchants.list_whitelisted_ips()

# Add a new IP
zenpays.merchants.add_ip_to_whitelist(
    "203.0.113.50",
    "Production Server"
)

# Remove an IP
zenpays.merchants.remove_ip_from_whitelist("ip_entry_id")
```

Two-Factor Authentication

Enable 2FA for additional security:

JAVASCRIPT

```javascript
// Setup 2FA
const setup = await zenpays.security.setup2FA()
console.log('Scan this QR code:', setup.qrCodeUrl)
console.log('Backup codes:', setup.backupCodes)

// Verify 2FA code
await zenpays.security.verify2FA({ code: '123456' })

// Check 2FA status
const status = await zenpays.security.is2FAEnabled()
console.log('2FA enabled:', status.enabled)
```

PYTHON

```python
# Setup 2FA
setup = zenpays.security.setup_two_factor()
print(f"Scan this QR code: {setup['qr_code_url']}")
print(f"Backup codes: {setup['backup_codes']}")

# Verify 2FA code
zenpays.security.verify_two_factor({"code": "123456"})

# Check 2FA status
status = zenpays.security.is_two_factor_enabled()
print(f"2FA enabled: {status['enabled']}")
```

Best Practices

1. **Use environment variables** - Never hardcode API keys
2. **Rotate keys regularly** - Create new keys and revoke old ones
3. **Use minimal scopes** - Only grant necessary permissions
4. **Enable IP whitelisting** - Restrict access to known IPs
5. **Enable 2FA** - Add an extra layer of security
6. **Monitor usage** - Check API key usage in the dashboard

Handling Authentication Errors

JAVASCRIPT

```javascript
import { AuthenticationError, AuthorizationError } from '@zenxdigitalholdings/zenpays'

try {
  await zenpays.payments.createPaymentIntent({
    amount: 1000,
    currency: 'USD',
  })
}
catch (error) {
  if (error instanceof AuthenticationError) {
    console.error('Invalid API key')
  }
  else if (error instanceof AuthorizationError) {
    console.error('API key lacks required permissions')
  }
}
```

PYTHON

```python
from zenpays.errors import AuthenticationError, PermissionError

try:
    zenpays.payments.create_payment_intent({
        "amount": 1000,
        "currency": "USD",
    })
except AuthenticationError:
    print("Invalid API key")
except PermissionError:
    print("API key lacks required permissions")
```

Next Steps

- Quick Start - Create your first payment
- Security Guide (https://docs.zenpays.com/docs/guides/security) - Learn more about security best practices
- Webhooks - Verify webhook signatures

PART

Guides

11 documents

GUIDES

Custom Checkout Integration

Build your own checkout page using ZenPays APIs. Instead of redirecting customers to the ZenPays hosted checkout, you can collect payment details on your own page and confirm the payment via API.

Flow Overview

```
1. Your Server          2. Your Checkout Page        3. Your Checkout Page        4. Customer Browser
   Create Intent  ──►      Fetch Intent Details  ──►    Confirm Payment     ──►    Redirect to TSP
   (server-side)           (client-side)                (client-side)               (payment page)
```

Step-by-step:

1. **Create Payment Intent** (server-side) — Your backend creates a payment intent via API with the amount, currency, and customer info.
2. **Fetch Payment Intent** (client-side) — Your checkout page fetches the intent details to get supported payment methods, countries, and channel limits.
3. **Confirm Payment** (client-side) — After the customer selects a payment method and fills in details, your page sends a confirm request.
4. **Redirect** — The confirm response returns a `redirectUrl`. Redirect the customer there to complete payment with the TSP (payment provider).
5. **Webhook** — ZenPays sends a webhook to your server when the payment succeeds or fails.

Base URL

```
https://api.zenpays.com/payment/api/v1
```

---

1. Create Payment Intent

Creates a payment intent on your server. This should be called from your **backend** (not the browser) since it requires your Merchant ID.

Request

```
POST /payment/api/v1/payment-intent
```

Headers:

Header	Required	Description
`X-Merchant-ID`	Yes	Your merchant ID
`Content-Type`	Yes	`application/json`
`X-Request-ID`	No	Unique request ID for idempotency

Body:

Field	Type	Required	Description
`amount`	number	Yes	Payment amount (minimum: 0.01)
`currency`	string	Yes	Currency code (e.g. `INR`, `USD`, `BRL`)
`customerEmail`	string	Yes*	Customer email (*required if no `customerId`)
`customerPhone`	string	No	Customer phone number
`customerFirstName`	string	No	Customer first name
`customerLastName`	string	No	Customer last name
`customerCountry`	string	No	ISO 3166-1 alpha-2 country code (e.g. `IN`)
`customerId`	string	No	Existing customer ID (skip email if provided)
`paymentMethod`	string	No	Preferred payment method
`description`	string	No	Payment description (max 500 chars)
`returnUrl`	string	No	URL to redirect customer after payment
`metadata`	object	No	Custom key-value data
`requestId`	string	No	Unique request ID (10-64 chars)

Example Request

```
{
  "amount": 500,
  "currency": "INR",
  "customerEmail": "customer@example.com",
  "customerPhone": "+911234567890",
  "customerFirstName": "John",
  "customerLastName": "Doe",
  "customerCountry": "IN",
  "description": "Order #12345",
  "returnUrl": "https://yoursite.com/payment-complete",
  "metadata": {
    "orderId": "order_12345"
  }
}
```

Example Response

```
{
  "success": true,
  "data": {
    "intentId": "pi_1775052514718_yn6v3nc47",
    "merchantId": "ZP_FIN_1774592804_0781001264",
    "amount": 500,
    "currency": "INR",
    "status": "requires_payment_method",
    "description": "Order #12345",
    "expiresAt": "2026-04-01T11:00:04.296Z",
    "processingFee": 0,
    "createdAt": "2026-04-01T10:50:04.299Z",
    "paymentPageUrl": "https://checkout.zenpays.com/pay/pi_1775052514718_yn6v3nc47?
token=pi_1775052514718_secret_abc123"
  },
  "message": "Payment intent created successfully",
  "error": null,
  "meta": {
    "request_id": "req_1775052514718_abc123",
    "timestamp": "2026-04-01T10:50:04.500Z",
    "processing_time_ms": 200,
    "api_version": "v1",
    "endpoint": "POST /payment-intent"
  }
}
```

Save the `intentId` — you'll need it for the next steps.

---

2. Fetch Payment Intent (Public)

Fetch the payment intent details on your checkout page. This is a **public endpoint** — no authentication required. Use this to render supported payment methods and validate amounts.

Request

```
GET /payment/api/v1/payment-intent/public/{intentId}
```

No headers required.

Example Response


```
{
  "success": true,
  "data": {
    "intentId": "pi_1775052514718_yn6v3nc47",
    "merchantId": "ZP_FIN_1774592804_0781001264",
    "merchantName": "Your Store",
    "merchantLogo": "https://cdn.example.com/logo.png",
    "amount": "500.00000000",
    "currency": "INR",
    "description": "Order #12345",
    "status": "requires_payment_method",
    "expiresAt": "2026-04-01T11:00:04.296Z",
    "isExpired": false,
    "timeRemaining": 598,
    "customerEmail": "customer@example.com",
    "customerPhone": "+911234567890",
    "supportedCountries": ["IN", "BR", "HK"],
    "supportedPaymentMethods": ["UPI", "UPIQR-H5", "BANK_TRANSFER"],
    "channelLimits": {
      "INR": {
        "UPI": { "min": 100, "max": 50000 },
        "UPIQR-H5": { "min": 500, "max": 50000 }
      }
    },
    "selectedTsp": "sulifu_pay",
    "processingFee": 0,
    "createdAt": "2026-04-01T10:50:04.299Z"
  },
  "message": "Payment intent details retrieved successfully",
  "error": null,
  "meta": {
    "request_id": "req_1775052515000_xyz789",
    "timestamp": "2026-04-01T10:50:05.000Z",
    "processing_time_ms": 372,
    "api_version": "v1",
    "endpoint": "GET /payment-intent/public/pi_1775052514718_yn6v3nc47",
    "user_type": "merchant"
  }
}
```

Key Fields for Your Checkout UI

Field	How to Use
`supportedPaymentMethods`	Render only these payment methods as options. Values like `"UPI"`, `"UPIQR-H5"`, `"BANK_TRANSFER"`, `"CARD"`, `"PIX"`.
`channelLimits`	Validate the payment amount against min/max per method. If the amount is outside the range, disable or hide that method.
`isExpired` / `timeRemaining`	If `isExpired` is `true`, show an error — the intent can no longer be paid. `timeRemaining` is in seconds.
`status`	Must be `"requires_payment_method"` to proceed. Any other status means the intent is already processing or completed.
`amount` / `currency`	Display the payment amount to the customer.
`merchantName` / `merchantLogo`	Display merchant branding on your checkout page.

Edge Cases to Handle

- **Expired intent**: If `isExpired` is `true`, show "Payment link expired" and don't allow confirmation.
- **Already paid**: If `status` is `"succeeded"` or `"processing"`, show the appropriate message.
- **Amount below channel limit**: If `channelLimits.INR.UPI.min` is 100 and the amount is 50, don't show UPI as an option.
- **UPI methods only for INR**: `UPI` and `UPIQR-H5` are only valid when currency is `INR`. Don't show them for other currencies.
- **No supported methods**: If `supportedPaymentMethods` is empty or none pass the channel limits filter, show an error.

---

3. Confirm Payment Intent

After the customer selects a payment method and fills in details, confirm the payment. This is a **public endpoint**.

Request

```
POST /payment/api/v1/payment-intent/{intentId}/confirm
```

Headers:

Header	Required	Description
`Content-Type`	Yes	`application/json`

Request Body Structure

```
{
  "customerDetails": {
    "name": "Customer Name",
    "email": "customer@example.com",
    "phone": "+911234567890",
    "address": {
      "country": "IN"
    }
  },
  "paymentMethodDetails": {
    "paymentMethod": "FIAT",
    "paymentType": "UPI"
  },
  "deviceInfo": {
    "userAgent": "Mozilla/5.0 ...",
    "browserInfo": "Mozilla/5.0 ...",
    "screenWidth": 1512,
    "screenHeight": 982,
    "deviceFingerprint": "unique_fingerprint_hash",
    "metadata": {
      "selectedPaymentMethod": "fiat",
      "selectedMethod": "upi",
      "specificPaymentMethod": "UPI",
      "customerCurrency": "INR"
    }
  },
  "confirmSource": "MERCHANT_SITE"
}
```

Field Reference

customerDetails (required):

Field	Type	Required	Description
`name`	string	Yes	Full customer name
`email`	string	Yes	Customer email
`phone`	string	No	Customer phone
`address.country`	string	No	ISO country code

paymentMethodDetails (required):

Field	Type	Description
`paymentMethod`	string	Always `"FIAT"` for fiat payments
`paymentType`	string	The payment type (see table below)

confirmSource (required for custom checkout):

Always set to `"MERCHANT_SITE"`. The value `"ZENPAY_CHECKOUT"` is reserved for the ZenPays hosted checkout page.

Payment Method Examples

UPI Payment

```
{
  "paymentMethodDetails": {
    "paymentMethod": "FIAT",
    "paymentType": "UPI",
    "upiId": ""
  }
}
```

The `upiId` field can be empty — the TSP payment page will collect it.

UPIQR-H5 Payment (QR Code)

```
{
  "paymentMethodDetails": {
    "paymentMethod": "FIAT",
    "paymentType": "UPIQR-H5"
  }
}
```

No additional fields needed. The TSP generates the QR code on their payment page.

Bank Transfer

```
{
  "paymentMethodDetails": {
    "paymentMethod": "FIAT",
    "paymentType": "BANK_TRANSFER",
    "bankCode": "dp_hdfcbank_in"
  }
}
```

Card Payment

```
{
  "paymentMethodDetails": {
    "paymentMethod": "FIAT",
    "paymentType": "CARD",
    "cardNumber": "4111111111111111",
    "expiryMonth": "12",
    "expiryYear": "2027",
    "cvv": "123",
    "cardHolderName": "John Doe"
  }
}
```

PIX (Brazil)

```
{
  "paymentMethodDetails": {
    "paymentMethod": "FIAT",
    "paymentType": "PIX"
  }
}
```

Payment Type Reference

paymentType	Currency	Extra Fields
`UPI`	INR	`upiId` (optional)
`UPIQR-H5`	INR	None
`BANK_TRANSFER`	INR	`bankCode` (optional)
`CARD`	Any	`cardNumber`, `expiryMonth`, `expiryYear`, `cvv`, `cardHolderName`
`PIX`	BRL	None
`NETBANKING`	INR	`bankCode` (optional)
`WALLET`	INR	`walletProvider` (optional)

Confirm Response

Example (UPI)

```
{
  "success": true,
  "data": {
    "intentId": "pi_1775052514718_yn6v3nc47",
    "redirectUrl": "https://cashier.ppco.lol?orderNo=C20260401194351420938 57",
    "tspProvider": "sulifu_pay",
    "externalTransactionId": "zpd6a82e78cedc1775052830530",
    "status": "created",
    "message": "Payment confirmed successfully. Redirecting to TSP.",
    "requiresCardDetails": false,
    "supportedCurrencies": ["INR"],
    "supportedCountries": ["IN", "BR", "HK"],
    "transactionId": "txn_1775052514718_1_mng4mhqh_p9axu5"
  },
  "message": "Payment confirmation successful",
  "error": null,
  "meta": {
    "request_id": "req_1775052830084_b0c0htr5e",
    "timestamp": "2026-04-01T14:13:51.654Z",
    "processing_time_ms": 1569,
    "api_version": "v1",
    "endpoint": "POST /payment-intent/pi_1775052514718_yn6v3nc47/confirm",
    "user_type": "merchant"
  }
}
```

What to Do with the Response

1. **Check** `success` — If `false`, show the error message to the customer.
2. **Redirect** — If `redirectUrl` is present, redirect the customer's browser to that URL. This is the TSP's payment page where the customer completes the payment (enters UPI PIN, scans QR code, etc.).
3. **Save** `transactionId` — Store this for tracking and reconciliation.

```
const response = await fetch(`/payment/api/v1/payment-intent/${intentId}/confirm`, {
  method: 'POST',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify(confirmPayload),
});

const result = await response.json();

if (result.success && result.data.redirectUrl) {
  // Redirect customer to TSP payment page
  window.location.href = result.data.redirectUrl;
} else {
  // Show error
  alert(result.data?.message || result.error?.message || 'Payment failed');
}
```

Edge Cases and Error Handling

Payment Intent Errors

Scenario	How to Detect	What to Show
Intent expired	`isExpired: true` from GET response	"This payment link has expired. Please request a new one."
Intent already paid	`status: "succeeded"`	"This payment has already been completed."
Intent processing	`status: "processing"`	"This payment is currently being processed."
Intent not found	404 from GET endpoint	"Payment not found."

Confirm Errors

Scenario	How to Detect	What to Show
Amount below minimum	Error response with min amount message	"Minimum amount for UPI is 100 INR."
Amount above maximum	Error response with max amount message	"Maximum amount for UPI is 50,000 INR."
Invalid payment method	400 error	"Selected payment method is not supported."
TSP unavailable	500 error with routing failure	"Payment service temporarily unavailable. Please try again."

Channel Limits Validation

Before showing payment methods to the customer, filter out methods where the amount is outside the channel limits:

```
const intentData = await fetchPaymentIntent(intentId);
const { amount, currency, supportedPaymentMethods, channelLimits } = intentData;

const currencyLimits = channelLimits?.[currency.toUpperCase()];

const availableMethods = supportedPaymentMethods.filter(method => {
  if (!currencyLimits) return true;
  const limit = currencyLimits[method];
  if (!limit) return true;
  return amount >= limit.min && amount <= limit.max;
});

// Render only `availableMethods` as options
```

UPI is INR-Only

`UPI` and `UPIQR-H5` payment methods are only valid when the currency is `INR`. Don't show them for other currencies:

```
const UPI_METHODS = ['UPI', 'UPIQR-H5'];
const isINR = currency.toUpperCase() === 'INR';

const filteredMethods = availableMethods.filter(method => {
  if (UPI_METHODS.includes(method) && !isINR) return false;
  return true;
});
```

Webhooks

After the customer completes payment on the TSP page, ZenPays sends a webhook to your configured endpoint with the payment result.

See the Webhooks Guide for setup instructions and payload format.

Full Integration Example

```
// 1. Server-side: Create payment intent
const createResponse = await fetch('https://api.zenpays.com/payment/api/v1/payment-intent', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json',
    'X-Merchant-ID': 'your_merchant_id',
  },
  body: JSON.stringify({
    amount: 500,
    currency: 'INR',
    customerEmail: 'customer@example.com',
    returnUrl: 'https://yoursite.com/payment-complete',
  }),
});
const { data: { intentId } } = await createResponse.json();

// 2. Client-side: Fetch intent details
const intentResponse = await fetch(
  `https://api.zenpays.com/payment/api/v1/payment-intent/public/${intentId}`
);
const { data: intent } = await intentResponse.json();

if (intent.isExpired) {
  showError('Payment link expired');
  return;
}

// 3. Client-side: Render payment methods filtered by channel limits
const methods = filterMethodsByLimits(
  intent.supportedPaymentMethods,
  intent.channelLimits,
  intent.currency,
  intent.amount
);

// 4. Client-side: After customer selects UPI and submits
const confirmResponse = await fetch(
  `https://api.zenpays.com/payment/api/v1/payment-intent/${intentId}/confirm`,
  {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({
      customerDetails: {
        name: 'John Doe',
        email: 'customer@example.com',
        phone: '+911234567890',
        address: { country: 'IN' },
      },
      paymentMethodDetails: {
        paymentMethod: 'FIAT',
        paymentType: 'UPI',
        upiId: '',
      },
      deviceInfo: {
        userAgent: navigator.userAgent,
        screenWidth: window.screen.width,
        screenHeight: window.screen.height,
      },
      confirmSource: 'MERCHANT_SITE',
    }),
  }
);
```

```
const result = await confirmResponse.json();

// 5. Redirect to TSP payment page
if (result.success && result.data.redirectUrl) {
  window.location.href = result.data.redirectUrl;
}
```

UPI Native Link Handling (INR Payments)

For INR payments, the `redirectUrl` opens a TSP payment page that shows a QR code. On mobile, customers can't scan a QR code on the same device they're paying from. ZenPays provides tools to **extract the `upi://` deep link** from the QR code so you can:

- **Mobile**: Open the customer's UPI app directly (GPay, PhonePe, Paytm) with one tap
- **Desktop**: Show your own clean QR code + "Copy UPI Link" button

Quick Start: Poll ZenPays API

After confirming payment, poll the ZenPays API to get the extracted UPI link:

```
// After confirm returns redirectUrl for INR payment:
if (result.data.redirectUrl && currency === 'INR') {
  const isMobile = /Mobi|Android|iPhone/i.test(navigator.userAgent);

  // Extract cashierOrderNo from redirectUrl for faster lookup
  let cashierOrderNo = '';
  try {
    cashierOrderNo = new URL(result.data.redirectUrl).searchParams.get('orderNo') || '';
  } catch {}

  // Poll every 4 seconds (ZenPays extracts the QR server-side)
  const interval = setInterval(async () => {
    const params = new URLSearchParams();
    if (cashierOrderNo) params.set('cashierOrderNo', cashierOrderNo);
    params.set('payPageUrl', result.data.redirectUrl);

    const res = await fetch(
      `https://api.zenpays.com/payment/api/v1/payment-intent/${intentId}/upi-link?${params}`
    );
    const data = await res.json();

    if (data.success && data.data?.upiDeepLink) {
      clearInterval(interval);
      const upiLink = data.data.upiDeepLink;

      if (isMobile && upiLink.startsWith('upi://')) {
        // Mobile: open UPI app with brief delay
        setTimeout(() => { window.location.href = upiLink; }, 600);
      } else {
        // Desktop: show your own QR + copy button (don't redirect)
        showDesktopUpiUI(upiLink);
      }
    }
  }, 4000);

  // Stop polling after 60 seconds
  setTimeout(() => clearInterval(interval), 60000);
}
```

Full Guide

For complete implementation with multiple approaches (server-side headless browser extraction, frontend screen capture), code examples in JavaScript, Python, and PHP, and best practices, see the **UPI Native Link Guide** (https://docs.zenpays.com/docs/guides/upi-native-link-guide).

GUIDES

Error Handling

Learn how to handle errors in the ZenPays SDK.

Error Types

The SDK provides specific error classes for different scenarios:

Error Class	HTTP Status	Description
`ZenPaysError`	Various	Base error class
`AuthenticationError`	401	Invalid API key
`AuthorizationError`	403	Insufficient permissions
`NotFoundError`	404	Resource not found
`ValidationError`	400	Invalid input data
`RateLimitError`	429	Rate limit exceeded
`NetworkError`	-	Network connectivity issues
`PaymentError`	402	Payment processing failed
`ConfigurationError`	-	SDK misconfiguration

Basic Error Handling

JAVASCRIPT

```javascript
import {
  AuthenticationError,
  NotFoundError,
  ValidationError,
  ZenPaysError,
} from '@zenxdigitalholdings/zenpays'

try {
  await zenpays.payments.createPaymentIntent({
    amount: 1000,
    currency: 'USD',
  })
}
catch (error) {
  if (error instanceof AuthenticationError) {
    console.error('Invalid API key. Check your credentials.')
  }
  else if (error instanceof ValidationError) {
    console.error('Validation failed:', error.message)
    console.error('Fields:', error.fields)
  }
  else if (error instanceof NotFoundError) {
    console.error('Resource not found')
  }
  else if (error instanceof ZenPaysError) {
    console.error(`API Error [${error.code}]: ${error.message}`)
    console.error('Status:', error.status)
    console.error('Details:', error.details)
  }
  else {
    throw error // Re-throw unexpected errors
  }
}
```

PYTHON

```python
from zenpays import ZenPays
from zenpays.errors import (
    AuthenticationError,
    NotFoundError,
    ValidationError,
    ZenPaysError,
)

try:
    zenpays.payments.create_payment_intent({
        "amount": 1000,
        "currency": "USD",
    })
except AuthenticationError:
    print("Invalid API key. Check your credentials.")
except ValidationError as e:
    print(f"Validation failed: {e}")
    print(f"Fields: {e.fields}")
except NotFoundError:
    print("Resource not found")
except ZenPaysError as e:
    print(f"API Error [{e.code}]: {e}")
    print(f"Status: {e.status}")
    print(f"Details: {e.details}")
except Exception as e:
    raise  # Re-raise unexpected errors
```

Error Properties

All ZenPays errors include:

```typescript
interface ZenPaysError extends Error {
  message: string // Human-readable message
  code?: string // Error code (e.g., 'VALIDATION_ERROR')
  status?: number // HTTP status code
  details?: string // Additional details
}
```

Rate Limiting

Handle rate limits gracefully:

JAVASCRIPT

```javascript
import { RateLimitError } from '@zenxdigitalholdings/zenpays'

async function makeRequestWithRetry(fn: () => Promise<any>, maxRetries = 3) {
  for (let i = 0; i < maxRetries; i++) {
    try {
      return await fn()
    }
    catch (error) {
      if (error instanceof RateLimitError && i < maxRetries - 1) {
        const delay = error.retryAfter ?? (2 ** i * 1000)
        console.log(`Rate limited. Retrying in ${delay}ms...`)
        await new Promise(resolve => setTimeout(resolve, delay))
      }
      else {
        throw error
      }
    }
  }
}

// Usage
const intent = await makeRequestWithRetry(() =>
  zenpays.payments.createPaymentIntent({ amount: 1000, currency: 'USD' })
)
```

PYTHON

```python
import time
from zenpays.errors import RateLimitError


def make_request_with_retry(fn, max_retries=3):
    for i in range(max_retries):
        try:
            return fn()
        except RateLimitError as e:
            if i < max_retries - 1:
                delay = e.retry_after if e.retry_after else (2 ** i * 1000) / 1000
                print(f"Rate limited. Retrying in {delay}s...")
                time.sleep(delay)
            else:
                raise


# Usage
intent = make_request_with_retry(
    lambda: zenpays.payments.create_payment_intent({"amount": 1000, "currency": "USD"})
)
```

Network Errors

Handle network connectivity issues:

JAVASCRIPT

```javascript
import { NetworkError } from '@zenxdigitalholdings/zenpays'

try {
  await zenpays.payments.createPaymentIntent({
    amount: 1000,
    currency: 'USD',
  })
}
catch (error) {
  if (error instanceof NetworkError) {
    if (error.message === 'Request timeout') {
      console.error('Request timed out. Try again later.')
    }
    else {
      console.error('Network error. Check your connection.')
    }
  }
}
```

PYTHON

```python
from zenpays.errors import NetworkError

try:
    zenpays.payments.create_payment_intent({
        "amount": 1000,
        "currency": "USD",
    })
except NetworkError as e:
    if str(e) == "Request timeout":
        print("Request timed out. Try again later.")
    else:
        print("Network error. Check your connection.")
```

Payment Errors

Handle payment-specific errors:

JAVASCRIPT

```javascript
import { PaymentError } from '@zenxdigitalholdings/zenpays'

try {
  await zenpays.payments.confirmPayment('pi_xxx', {
    customerDetails: { ... },
    paymentMethodDetails: { ... },
  })
} catch (error) {
  if (error instanceof PaymentError) {
    console.error('Payment failed:', error.message)
    console.error('Intent ID:', error.paymentIntentId)
    // Show user-friendly message
  }
}
```

PYTHON

```python
from zenpays.errors import PaymentError

try:
    zenpays.payments.confirm_payment("pi_xxx", {
        "customer_details": { ... },
        "payment_method_details": { ... },
    })
except PaymentError as e:
    print(f"Payment failed: {e}")
    print(f"Intent ID: {e.payment_intent_id}")
    # Show user-friendly message
```

Validation Errors

Handle validation errors with field-level details:

JAVASCRIPT

```javascript
import { ValidationError } from '@zenxdigitalholdings/zenpays'

try {
  await zenpays.customers.create({
    email: 'invalid-email',
    name: '',
  })
}
catch (error) {
  if (error instanceof ValidationError) {
    console.error('Validation errors:')
    if (error.fields) {
      for (const [field, messages] of Object.entries(error.fields)) {
        console.error(`  ${field}: ${messages.join(', ')}`)
      }
    }
  }
}
```

PYTHON

```python
from zenpays.errors import ValidationError

try:
    zenpays.customers.create({
        "email": "invalid-email",
        "name": "",
    })
except ValidationError as e:
    print("Validation errors:")
    if e.fields:
        for field, messages in e.fields.items():
            print(f"  {field}: {', '.join(messages)}")
```

Channel Limit Errors

Handle payment method amount limit violations:

JAVASCRIPT

```javascript
import { ChannelLimitError } from '@zenxdigitalholdings/zenpays'

try {
  await zenpays.payments.confirmPayment('pi_xxx', {
    customerDetails: { ... },
    paymentMethodDetails: { ... },
  })
} catch (error) {
  if (error instanceof ChannelLimitError) {
    const { amount, currency, paymentMethod, minimumAmount, maximumAmount } = error.limits ?? {}
    console.error(`Amount ${amount} ${currency} is out of range for ${paymentMethod}`)
    console.error(`Allowed range: ${minimumAmount} - ${maximumAmount} ${currency}`)
    // Prompt user to adjust the amount or select a different payment method
  }
}
```

PYTHON

```python
from zenpays.errors import ChannelLimitError

try:
    zenpays.payments.confirm_payment("pi_xxx", {
        "customer_details": { ... },
        "payment_method_details": { ... },
    })
except ChannelLimitError as e:
    limits = e.limits or {}
    print(f"Amount {limits.get('amount')} {limits.get('currency')} is out of range for
{limits.get('paymentMethod')}")
    print(f"Allowed range: {limits.get('minimumAmount')} - {limits.get('maximumAmount')}
{limits.get('currency')}")
    # Prompt user to adjust the amount or select a different payment method
```

Best Practices

1. **Always catch errors** - Don't let errors crash your application
2. **Use specific error types** - Check for specific errors before generic ones
3. **Log error details** - Include code, status, and details for debugging
4. **Show user-friendly messages** - Don't expose technical details to users
5. **Implement retry logic** - Handle transient errors automatically
6. **Monitor errors** - Track error rates in production

Example: Complete Error Handler

JAVASCRIPT

```
import {
  AuthenticationError,
  AuthorizationError,
  ChannelLimitError,
  NetworkError,
  NotFoundError,
  PaymentError,
  RateLimitError,
  ValidationError,
  ZenPaysError,
} from '@zenxdigitalholdings/zenpays'

function handleZenPaysError(error: unknown): {
  message: string
  shouldRetry: boolean
  retryDelay?: number
} {
  if (error instanceof AuthenticationError) {
    return { message: 'Invalid API key', shouldRetry: false }
  }

  if (error instanceof AuthorizationError) {
    return { message: 'Access denied', shouldRetry: false }
  }

  if (error instanceof NotFoundError) {
    return { message: 'Resource not found', shouldRetry: false }
  }

  if (error instanceof ValidationError) {
    return { message: error.message, shouldRetry: false }
  }

  if (error instanceof ChannelLimitError) {
    return { message: error.message, shouldRetry: false }
  }

  if (error instanceof RateLimitError) {
    return {
      message: 'Too many requests',
      shouldRetry: true,
      retryDelay: error.retryAfter ?? 5000,
    }
  }

  if (error instanceof NetworkError) {
    return { message: 'Network error', shouldRetry: true, retryDelay: 1000 }
  }

  if (error instanceof PaymentError) {
    return { message: 'Payment failed', shouldRetry: false }
  }

  if (error instanceof ZenPaysError) {
    return { message: error.message, shouldRetry: false }
  }

  return { message: 'An unexpected error occurred', shouldRetry: false }
}
```

PYTHON

```python
from typing import TypedDict, Optional
from zenpays.errors import (
    AuthenticationError,
    AuthorizationError,
    ChannelLimitError,
    NetworkError,
    NotFoundError,
    PaymentError,
    RateLimitError,
    ValidationError,
    ZenPaysError,
)


class ErrorResult(TypedDict):
    message: str
    should_retry: bool
    retry_delay: Optional[int]


def handle_zenpays_error(error: Exception) -> ErrorResult:
    if isinstance(error, AuthenticationError):
        return {"message": "Invalid API key", "should_retry": False, "retry_delay": None}

    if isinstance(error, AuthorizationError):
        return {"message": "Access denied", "should_retry": False, "retry_delay": None}

    if isinstance(error, NotFoundError):
        return {"message": "Resource not found", "should_retry": False, "retry_delay": None}

    if isinstance(error, ValidationError):
        return {"message": str(error), "should_retry": False, "retry_delay": None}

    if isinstance(error, ChannelLimitError):
        return {"message": str(error), "should_retry": False, "retry_delay": None}

    if isinstance(error, RateLimitError):
        return {
            "message": "Too many requests",
            "should_retry": True,
            "retry_delay": error.retry_after if hasattr(error, "retry_after") and
error.retry_after else 5000,
        }

    if isinstance(error, NetworkError):
        return {"message": "Network error", "should_retry": True, "retry_delay": 1000}

    if isinstance(error, PaymentError):
        return {"message": "Payment failed", "should_retry": False, "retry_delay": None}

    if isinstance(error, ZenPaysError):
        return {"message": str(error), "should_retry": False, "retry_delay": None}

    return {"message": "An unexpected error occurred", "should_retry": False, "retry_delay": None}
```

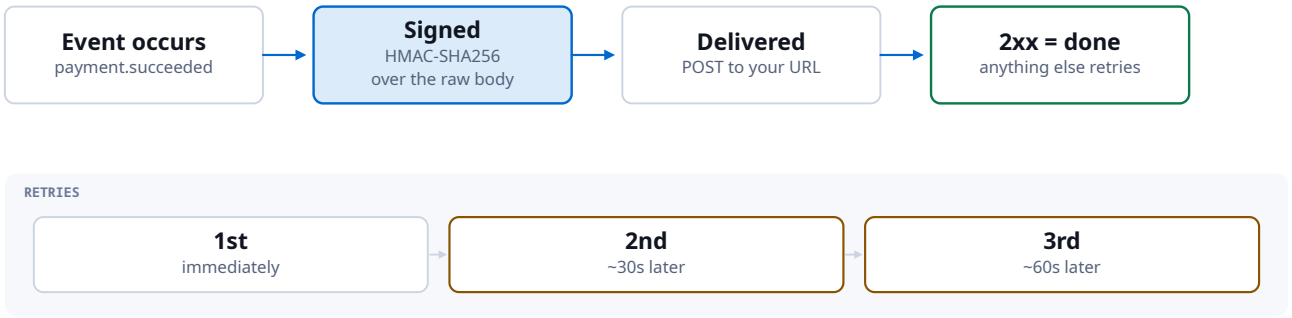
Webhooks

GUIDES / WEBHOOKS

Webhooks Overview

Receive real-time notifications about payment events.

Every webhook is signed over the exact bytes we send, and retried up to three times.

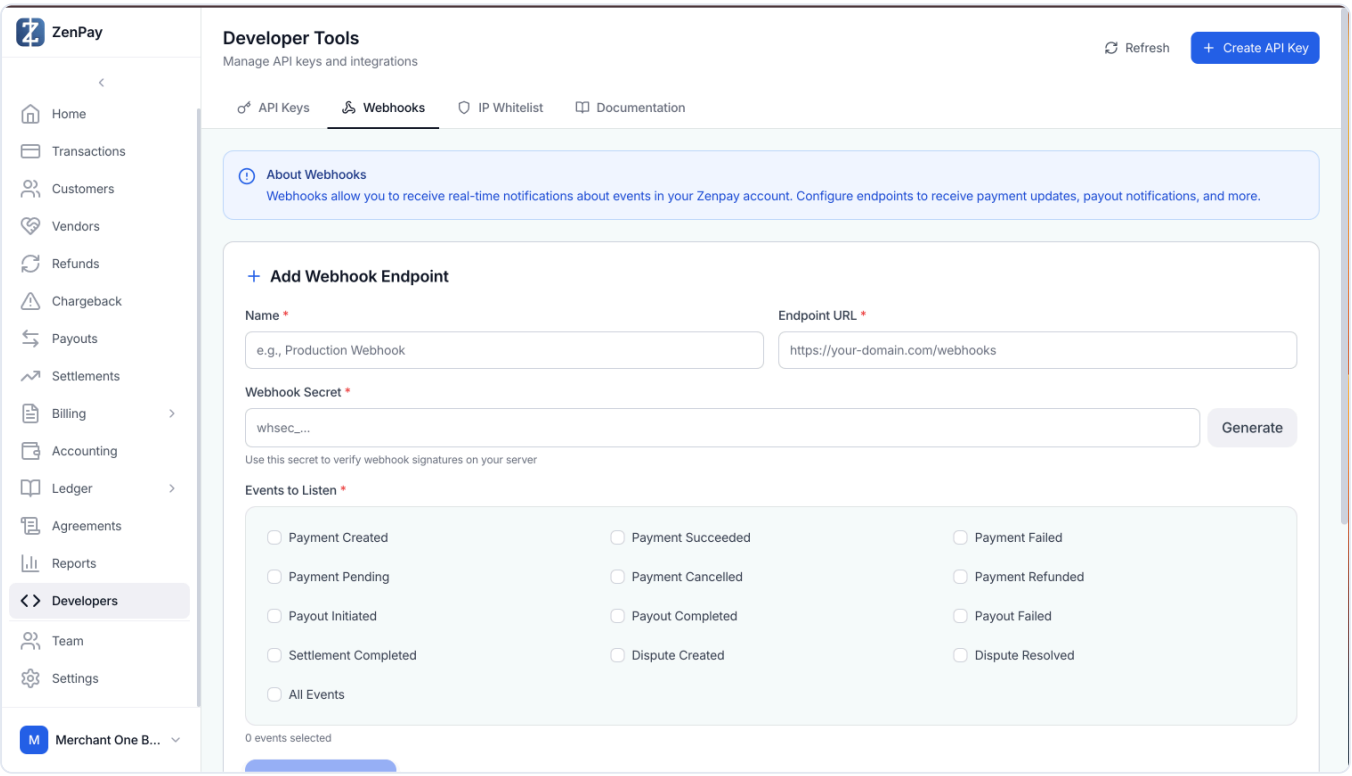


After three failed attempts delivery stops. Replay it from the dashboard once your endpoint is healthy.
Verify the signature against the raw request body, before any JSON parsing or re-serialising.

How webhooks are signed, delivered and retried

Creating a Webhook

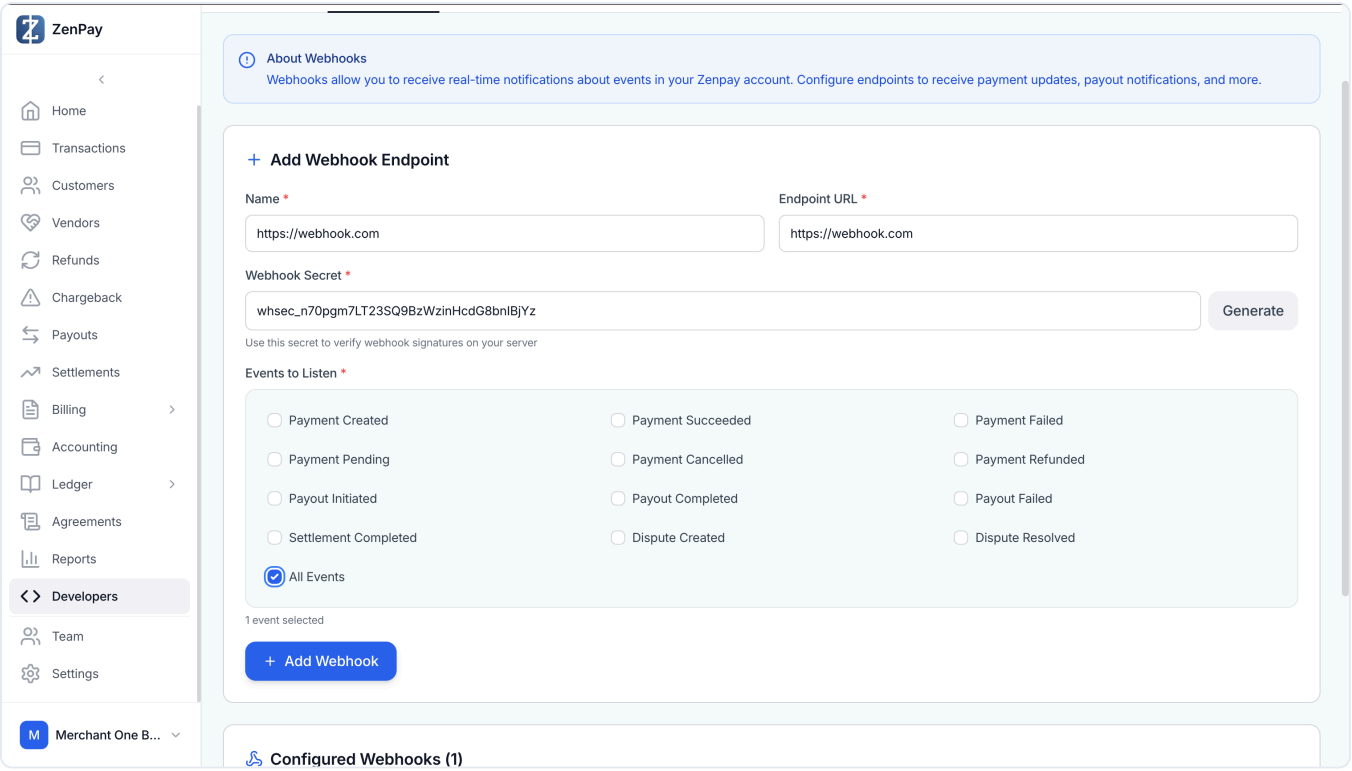
1. Log in to your **Merchant Dashboard** and navigate to **Developer Tools → Webhooks** tab.



2. Fill in the **Name** and **Endpoint URL** fields with your server's webhook URL (e.g. `https://your-domain.com/webhooks/zenpays`).
3. Click **Generate** to create a **Webhook Secret**. Copy and store this secret securely on your server — you'll need it to verify webhook signatures.

4. Select the **Events to Listen** to. You can pick individual events or select **All Events** to receive everything.

5. Click **+ Add Webhook** to save.



6. Your endpoint will appear under **Configured Webhooks**. You can add multiple webhook endpoints to receive events at different URLs.

Testing Webhooks

To test your webhook integration:

1. Create a webhook on the dashboard (see above).
2. Process a test payment transaction through the checkout flow.
3. Check your server logs to verify the webhook was received with the correct payload and signature.

Delivery attempts and status are visible in the dashboard under each configured webhook endpoint.

GUIDES / WEBHOOKS

Payment Webhooks

Payment webhook events track the lifecycle of a payment from initiation to completion or failure.

Events

Event	Description
`payment.success`	Payment completed successfully
`payment.failed`	Payment failed
`payment.pending`	Payment is pending
`payment.cancelled`	Payment was cancelled

Payload

```
{
  "event_type": "payment.success",
  "payment_data": {
    "merchant_id": "m_abc123",
    "payment_id": "pi_xyz789",
    "transaction_id": "txn_def456",
    "order_id": "pi_xyz789",
    "amount": 100.00,
    "currency": "USD",
    "status": "succeeded",
    "payment_method": "credit_card",
    "processed_at": "2026-02-28T10:30:00.000Z"
  }
}
```

For failed payments, an additional `failure_reason` field is included in `payment_data` along with an operational `code` (see Operational Errors (https://docs.zenpays.com/docs/rest-api/errors#operational-errors)) that maps to a friendly customer-facing message.

```
{
  "event_type": "payment.failed",
  "payment_data": {
    "merchant_id": "m_abc123",
    "payment_id": "pi_xyz789",
    "transaction_id": "txn_def456",
    "amount": 100.00,
    "currency": "USD",
    "status": "failed",
    "code": "TSP_PAYMENT_FAILED",
    "failure_reason": "Card declined by issuer",
    "processed_at": "2026-02-28T10:30:00.000Z"
  }
}
```

Payment Callback Lifecycle

Treat payment events as a state machine:

- `payment.pending` means the payment is still in progress.
- `payment.success` is the final success signal.
- `payment.failed` is an explicit final failure signal.

Incomplete payments can remain pending

Some payment channels keep incomplete orders in `processing` / `pending` until a terminal success is received. In these cases, your server may receive repeated or unchanged pending updates and may not receive an automatic failure callback.

Your integration should:

- keep the order as pending while waiting for a terminal event,
- mark success only on `payment.success`,
- mark failed only when `payment.failed` is explicitly delivered.

Recommended merchant handling

If no terminal callback arrives yet, continue showing the payment as pending/in progress. Do not auto-fail based only on timeout unless your own business timeout policy requires it.

Real-time WebSocket events

If you embed the ZenPays checkout (or subscribe to its `/payment-status` namespace directly), you'll also see these socket events alongside the HTTP webhooks above. They drive the live status UI in the customer's browser.

Event	When fired	Notable fields
`payment-update`	Generic status transition	`status`, optionally `rampOrderStatus`, `kycStatus`
`payment-success`	Intent moved to `succeeded`	`transactionId`, `timestamp`
`payment-failed`	Intent moved to terminal failed state	`code`, `error`, `timestamp` — the `code` is one of the operational error codes (https://docs.zenpays.com/docs/rest-api/errors#operational-errors) (e.g., `TSP_PAYMENT_FAILED`, `PROVIDER_UNAVAILABLE`, `PAYMENT_STUCK`). Use it to pick a friendly message; never display the raw `error` string.
`payment-stuck`	Intent has been in `processing` past ~90s with no transitions	`code`, `message`, `status: 'processing'` — **informational only**. The intent is still alive. Use this to swap a generic spinner for a "this is taking longer than usual" message; do NOT treat it as a terminal failure.

Example: handling `payment-failed` with codes

```
socket.on('payment-failed', (data: { intentId: string; code?: string; error?: string }) => {
  switch (data.code) {
    case 'NO_TSP_AVAILABLE':
    case 'UNSUPPORTED_REGION':
      showMessage("This payment route isn't available right now.");
      break;
    case 'UNSUPPORTED_CURRENCY_PAIR':
      showMessage("This currency combination isn't supported.");
      break;
    case 'TSP_PAYMENT_FAILED':
      showMessage("Your payment couldn't be completed. Please try a different method.");
      break;
    case 'PAYMENT_STUCK':
      // Hard timeout — friendly but explicit
      showMessage("We tried but couldn't complete this payment. Please try again later.");
      break;
    default:
      showMessage("Something went wrong. Please try again or contact support.");
  }
});
```

Example: handling `payment-stuck`

```
socket.on('payment-stuck', (data: { intentId: string; code: string; message: string }) => {
  // Swap the generic spinner for a friendlier "still working" UI.
  // Do NOT mark the payment as failed — it's still in progress.
  showWaitingCard({
    title: "This is taking longer than usual",
    description: "We're still trying. Feel free to wait or check back in your dashboard.",
  });
});
```

GUIDES / WEBHOOKS

Payout Webhooks

Payout webhook events track single payouts, payout intents, and batch payouts.

Single Payout Events

Event	Description
`payout.initiated`	Payout accepted by ZenPay and moved into processing (funds are reserved but the final outcome is not yet known)
`payout.completed`	Payout processing finished successfully and funds have been fully debited from the reserved balance
`payout.failed`	Payout processing did not complete successfully and any reserved funds are released or returned according to ZenPay rules
`payout.cancelled`	Payout was cancelled before processing completed

Payload

```
{
  "event_type": "payout.completed",
  "payment_data": {
    "payoutId": "payout_1774778823937_17298d89",
    "transactionId": "po6efbc8502da51774842226226",
    "intentId": "poi_mnblh1o5_e8190e32",
    "amount": 100,
    "currency": "INR",
    "status": "completed",
    "beneficiary": "Rajesh Kumar",
    "payoutType": "BANK_TRANSFER",
    "failureReason": null,
    "metadata": {
      "order_no": "ORD-2026-001",
      "customer_ref": "CUST-789"
    }
  }
}
```

Payload Fields

Field	Type	Description
`payoutId`	string	Internal payout ID
`transactionId`	string	External payout transaction reference — matches `externalPayoutId` from the confirm response (https://docs.zenpays.com/docs/rest-api/endpoints/payout-intents/confirm-payout-intent)
`intentId`	string | null	Payout intent ID (if created via the two-step intent flow)
`amount`	number	Payout amount

Field	Type	Description
`currency`	string	Payout currency
`status`	string	`initiated`, `completed`, `failed`, or `cancelled`
`beneficiary`	string	Beneficiary name
`payoutType`	string	Payout method (e.g. `BANK_TRANSFER`, `UPI`)
`failureReason`	string | null	Reason for failure (only on `payout.failed`)
`metadata`	object | null	Custom key-value pairs passed when creating the payout intent — returned as-is in all webhook callbacks

Linking payouts end-to-end

When using the two-step payout intent flow:

1. **Create intent** → get `intentId`
2. **Confirm intent** → get `intentId` + `payoutId` + `externalPayoutId`
3. **Webhook callback** → get `intentId` + `payoutId` + `transactionId`

The `transactionId` in the webhook equals `externalPayoutId` from the confirm response. All three IDs are present in both the confirm response and webhook, so you can always link them.

Payout Intent Events

Event	Description
`payout_intent.created`	Payout intent created
`payout_intent.confirmed`	Payout intent confirmed and submitted for processing
`payout_intent.succeeded`	Payout intent completed successfully
`payout_intent.failed`	Payout intent failed
`payout_intent.cancelled`	Payout intent cancelled
`payout_intent.expired`	Payout intent expired without confirmation

Payload

Shares the same payload shape as single payout webhooks:

```
{
  "event_type": "payout_intent.succeeded",
  "payment_data": {
    "intentId": "poi_mnblh1o5_e8190e32",
    "payoutId": "payout_1774778823937_17298d89",
    "transactionId": "po6efbc8502da51774842226226",
    "amount": 500,
    "currency": "INR",
    "status": "succeeded",
    "beneficiary": "Rajesh Kumar",
    "payoutType": "BANK_TRANSFER",
    "failureReason": null,
    "metadata": {
      "order_no": "ORD-2026-001"
    }
  }
}
```

Batch Payout Events

Event	Description
`batch_payout.created`	Batch payout created
`batch_payout.processing`	Batch payout is being processed
`batch_payout.completed`	All payouts in the batch completed successfully
`batch_payout.partially_completed`	Some payouts in the batch completed, others failed
`batch_payout.failed`	Batch payout failed
`batch_payout.cancelled`	Batch payout was cancelled

Payload

```
{
  "event_type": "batch_payout.completed",
  "payment_data": {
    "batchId": "bp_abc123",
    "totalPayouts": 50,
    "successCount": 48,
    "failedCount": 2,
    "totalAmount": 25000,
    "currency": "INR",
    "status": "completed",
    "completedAt": "2026-03-16T14:30:00.000Z"
  }
}
```

Payload Fields

Field	Type	Description
`batchId`	string	Batch payout ID
`totalPayouts`	number	Total payouts in the batch
`successCount`	number	Number of successful payouts
`failedCount`	number	Number of failed payouts
`totalAmount`	number	Total amount across all payouts
`currency`	string	Batch currency
`status`	string	`created`, `processing`, `completed`, `partially_completed`, `failed`, `cancelled`
`completedAt`	string | null	ISO 8601 completion timestamp

GUIDES / WEBHOOKS

Ramp Webhooks

Ramp webhook events notify you when a ramp intent (buy or sell) reaches a terminal state.

Events

Event	Description
`ramp.completed`	Ramp intent completed (buy: crypto sent to wallet, sell: fiat payout submitted)
`ramp.failed`	Ramp intent failed (crypto send or payout submission failed)

Payloads

The payload varies depending on the ramp direction (buy vs sell) and outcome.

Buy Ramp Completed

Crypto has been sent to the customer's wallet:

```
{
  "event_type": "ramp.completed",
  "payment_data": {
    "merchant_id": "m_abc123",
    "intentId": "ri_1710345678000_a1b2c3d4e5f6g7h8",
    "type": "buy",
    "status": "completed",
    "fiatCurrency": "USD",
    "cryptoCurrency": "eth_ethereum",
    "amount": 100,
    "chain": "ethereum",
    "destinationAddress": "0x742d35Cc6634C0532925a3b844Bc9e7595f2bD28",
    "fireblocksTxId": "fb_tx_9a8b7c6d5e4f",
    "completedAt": "2026-03-16T14:30:00.000Z"
  }
}
```

Sell Ramp Completed

Fiat payout has been submitted to the customer's bank account:

```
{
  "event_type": "ramp.completed",
  "payment_data": {
    "merchant_id": "m_abc123",
    "intentId": "ri_1710345678000_a1b2c3d4e5f6g7h8",
    "type": "sell",
    "status": "completed",
    "fiatCurrency": "USD",
    "cryptoCurrency": "usdt_tron",
    "amount": 20,
    "payoutId": "po_xyz789",
    "payoutAmount": 19.50,
    "payoutCurrency": "USD",
    "completedAt": "2026-03-16T14:30:00.000Z"
  }
}
```

Ramp Failed

Buy or sell flow encountered an error:

```
{
  "event_type": "ramp.failed",
  "payment_data": {
    "merchant_id": "m_abc123",
    "intentId": "ri_1710345678000_a1b2c3d4e5f6g7h8",
    "type": "buy",
    "status": "failed",
    "fiatCurrency": "USD",
    "cryptoCurrency": "eth_ethereum",
    "amount": 100,
    "reason": "Crypto send failed",
    "failedAt": "2026-03-16T14:30:00.000Z"
  }
}
```

Payload Fields

Field	Type	Present	Description
`merchant_id`	string	Always	Your merchant ID
`intentId`	string	Always	Ramp intent ID (`ri_...`)
`type`	string	Always	`"buy"` or `"sell"`
`status`	string	Always	`"completed"` or `"failed"`
`fiatCurrency`	string	Always	ISO currency code (e.g. `USD` , `EUR`)
`cryptoCurrency`	string	Always	Crypto identifier (e.g. `eth_ethereum` , `usdt_tron`)
`amount`	number	Always	Original intent amount
`completedAt`	string	On success	ISO 8601 completion timestamp

Field	Type	Present	Description
`failedAt`	string	On failure	ISO 8601 failure timestamp
`reason`	string	On failure	Human-readable failure reason
`chain`	string	Buy success	Blockchain network (e.g. `ethereum`, `tron`)
`destinationAddress`	string	Buy success	Wallet address crypto was sent to
`cryptoTxId`	string	Buy success	Crypto transaction reference
`payoutId`	string	Sell success	Payout reference ID
`payoutAmount`	number	Sell success	Fiat amount paid out
`payoutCurrency`	string	Sell success	Fiat currency of payout

GUIDES / WEBHOOKS

KYC Webhooks

KYC webhook events notify you when a customer's ZenPays KYC verification status changes. These are fired for customers going through identity verification in the ramp checkout widget.

Events

Event	Description
`kyc.approved`	Customer's identity verification passed
`kyc.declined`	Customer's identity verification was rejected
`kyc.in_review`	Verification flagged for manual review
`kyc.expired`	Previous verification has expired

> **Note**
> Intermediate states like `pending`, `in_progress`, and `resubmitted` are **not** delivered as webhooks — they would be noise for most integrations. If you need that granularity, poll `GET /ramp-intents/:id/kyc-status` (https://docs.zenpays.com/docs/rest-api/endpoints/ramp/get-kyc-status) instead.

Payload

```
{
  "event_type": "kyc.approved",
  "user_id": "usr_abc123",
  "kyc_status": "approved",
  "aml_status": "clear",
  "decline_reason": null,
  "merchant_id": "mer_abc123",
  "affected_intent_ids": ["ri_1710345678000_a1b2c3d4e5f6g7h8"],
  "didit_session_id": "8a9b2c1d-e3f4-...",
  "occurred_at": "2026-03-16T14:30:00.000Z"
}
```

Payload Fields

Field	Type	Description
`event_type`	string	The KYC event: `kyc.approved`, `kyc.declined`, `kyc.in_review`, or `kyc.expired`
`user_id`	string	The `userId` you passed when creating the ramp intent
`kyc_status`	string	Verification state: `approved`, `declined`, `in_review`, or `expired`
`aml_status`	string | null	AML screening result: `clear`, `flagged`, `rejected`, `pending_review`, or `null` if not yet evaluated
`decline_reason`	string | null	Reason code when terminal: `kyc_rejected`, `kyc_expired`, or a provider-specific string. `null` for non-decline events.
`merchant_id`	string	Your merchant ID
`affected_intent_ids`	string[]	Ramp intents that this status change applied to. **May be empty** on late approvals — see "Multi-merchant fan-out & late approval" below.
`didit_session_id`	string | null	The underlying Didit verification session ID, for cross-referencing with our support team.
`occurred_at`	string	ISO 8601 timestamp of the status change

Multi-merchant fan-out & late approval

Two behaviors that surprise integrators if you don't model them:

1. **Multi-merchant fan-out.** A single customer can have ramp intents across multiple merchants. When their KYC status changes, every merchant that has any intent for that user receives the event independently. Each receives only the `affected_intent_ids` belonging to *their* merchant.
2. **Late approval after a prior decline.** If a user is declined today and approved tomorrow (e.g., after manual re-review), the original failed intents stay terminally `cancelled` — we do **not** auto-resume them. The merchant still receives `kyc.approved`, but with `affected_intent_ids: []`. Treat this as an instruction to update your **user verification cache** so future intents proceed without re-KYC, not as an event about a specific transaction.

Usage

KYC webhooks are useful for:

- **Updating user records** — mark a user as KYC-verified in your system when `kyc.approved` is received, so future ramp intents can pass `kycVerified: true`.

- **Handling rejections** — notify the customer or trigger support flows on `kyc.declined`. The `decline_reason` field tells you whether it was a hard rejection (`kyc_rejected`) or a session expiry (`kyc_expired`).
- **Monitoring review queues** — track `kyc.in_review` events for compliance reporting; pair with `aml_status` to know whether identity was approved but AML flagged for review.

Example

```
function handleWebhookEvent(event: any) {
  switch (event.event_type) {
    case 'kyc.approved':
      // Mark user as KYC verified in your database for future ramp intents
      await updateUserKycStatus(event.user_id, 'verified')

      if (event.affected_intent_ids.length > 0) {
        // The user had active intents that just unblocked — surface success
        // in any UI showing those intents.
        await notifyIntentsUnblocked(event.affected_intent_ids)
      } else {
        // Late approval after a prior decline — no active intent to resume.
        // Merchant may want to invite the customer to start a new transaction.
        await inviteRetry(event.user_id)
      }
      break

    case 'kyc.declined':
      // Hard rejection — notify the customer, offer support
      await notifyCustomer(event.user_id, 'KYC verification was not successful', {
        reason: event.decline_reason,
        affectedIntents: event.affected_intent_ids,
      })
      break

    case 'kyc.in_review':
      // Identity OK but flagged for review (often AML-driven)
      await flagForReview(event.user_id, {
        amlStatus: event.aml_status,
        affectedIntents: event.affected_intent_ids,
      })
      break

    case 'kyc.expired':
      // User will need to re-verify on the next ramp intent
      await updateUserKycStatus(event.user_id, 'expired')
      break
  }
}
```

Skip KYC on repeat transactions

Once you receive `kyc.approved` for a user, store that status in your system. For subsequent ramp intents, pass `kycVerified: true` so the customer doesn't have to verify again.

GUIDES / WEBHOOKS

Other Webhooks

Additional webhook events for refunds, settlements, chargebacks, and deposits.

Refund Events

Event	Description
`refund.initiated`	Refund initiated
`refund.completed`	Refund completed
`refund.failed`	Refund failed

Settlement Events

Event	Description
`settlement.initiated`	Settlement started
`settlement.completed`	Settlement completed
`settlement.failed`	Settlement failed

Chargeback Events

Event	Description
`chargeback.initiated`	Chargeback initiated
`chargeback.resolved`	Chargeback resolved

Deposit Events

Event	Description
`deposit.pending`	Crypto deposit pending
`deposit.confirmed`	Crypto deposit confirmed on-chain
`deposit.cleared`	Crypto deposit fully cleared

GUIDES / WEBHOOKS

Signature Verification

ZenPay signs every webhook delivery so you can verify it came from ZenPay and was not tampered with.

How Signatures Work

1. **Algorithm:** HMAC-SHA256 by default (configurable per endpoint to `sha256` or `sha512` in the dashboard).
2. **Signed content:** The raw JSON request body as a UTF-8 string.
3. **Signature format:** `{algorithm}={hex_digest}` — e.g. `sha256=a1b2c3d4e5f6...`
4. **Header:** `X-ZenPay-Signature` (default, configurable per endpoint).
5. **Secret:** Your webhook endpoint secret from the dashboard.

> **Use the raw body**
> Always verify against the **raw request body** (bytes/string), not a parsed-then-re-serialized JSON object. Re-serialization can change key ordering or whitespace, which invalidates the signature.

Webhook Headers

Every webhook delivery includes the following headers:

Header	Description	Example
`X-ZenPay-Signature`	`{algorithm}={hex_hmac_digest}` of the raw JSON body using your webhook secret	`sha256=a1b2c3d4...`
`X-Zenpay-Event`	Event type string	`payment.success`
`X-Zenpay-Event-Id`	Unique event ID for deduplication	`evt_abc123`
`X-Zenpay-Timestamp`	ISO 8601 delivery timestamp	`2026-02-28T10:30:45.123Z`
`X-Zenpay-Attempt`	Delivery attempt number (starts at 1)	`1`
`User-Agent`	Delivery user agent	`Zenpay-Webhook/1.0`

Verification Implementation

JAVASCRIPT / NODE.JS

Use the helpers shipped in the SDK — you do not need to implement HMAC yourself. Both are exported at the package root and take no API key, so they run inside a webhook route with no client configured:

```
import { constructWebhookEvent, verifyWebhookSignature } from '@zenxdigitalholdings/zenpays'

// Returns a boolean.
const ok = await verifyWebhookSignature(rawBody, signatureHeader, secret)

// Or: verify and parse in one step. Throws Error('Invalid webhook signature')
// on failure, so a successful return guarantees authenticity.
const event = await constructWebhookEvent<PaymentSuccessEvent>(rawBody, signatureHeader, secret)
```

Parameter	Type	Description
`rawBody`	`string`	The raw request body, exactly as received
`signatureHeader`	`string | undefined | null`	`X-ZenPay-Signature` value. Accepts `sha256=…` or bare hex
`secret`	`string`	Your webhook signing secret

Both are `async` (they use Web Crypto, with a `node:crypto` fallback below Node 18) and both return `false` / throw rather than error out on a missing body, header, or secret.

> **SHA-256 only**
> The bundled helpers implement HMAC-SHA256. If you have configured an endpoint to sign with `sha512`, verify it with the manual implementation below instead.

Full Express example:

```
import { constructWebhookEvent } from '@zenxdigitalholdings/zenpays'
import express from 'express'

const app = express()

app.post(
  '/webhooks/zenpays',
  // express.raw() keeps the body unparsed — express.json() would re-serialize
  // it and break the signature.
  express.raw({ type: 'application/json' }),
  async (req, res) => {
    const signature = req.headers['x-zenpay-signature'] as string
    const timestamp = req.headers['x-zenpay-timestamp'] as string
    const eventId = req.headers['x-zenpay-event-id'] as string

    // 1. Verify the signature and parse in one step.
    let event: unknown
    try {
      event = await constructWebhookEvent(
        req.body.toString(),
        signature,
        process.env.WEBHOOK_SECRET!,
      )
    }
    catch {
      return res.status(401).json({ error: 'Invalid signature' })
    }

    // 2. Check timestamp freshness (prevent replay attacks)
    const deliveryTime = new Date(timestamp)
    const now = new Date()
    if (Math.abs(now.getTime() - deliveryTime.getTime()) > 5 * 60 * 1000) {
      return res.status(401).json({ error: 'Webhook timestamp expired' })
    }

    // 3. Deduplicate using event ID
    // if (await isEventAlreadyProcessed(eventId)) {
    //   return res.status(200).send('OK')
    // }

    // 4. Process
    handleWebhookEvent(event)

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

Manual implementation (no SDK, or
sha512
endpoints)

```
import crypto from 'node:crypto'

function verifyWebhookSignature(
  rawBody: Buffer | string,
  signature: string,
  secret: string
): boolean {
  const [algorithm, receivedHash] = signature.split('=')

  const expectedHash = crypto
    .createHmac(algorithm, secret)
    .update(rawBody)
    .digest('hex')

  // Timing-safe comparison to prevent timing attacks
  const received = Buffer.from(receivedHash, 'hex')
  const expected = Buffer.from(expectedHash, 'hex')

  if (received.length !== expected.length) return false
  return crypto.timingSafeEqual(received, expected)
}
```

PYTHON

```python
import hmac
import hashlib
from datetime import datetime, timezone, timedelta
from flask import Flask, request, jsonify

app = Flask(__name__)

def verify_webhook_signature(raw_body: bytes, signature: str, secret: str) -> bool:
    algorithm, received_hash = signature.split("=", 1)

    expected_hash = hmac.new(
        secret.encode("utf-8"),
        raw_body,
        getattr(hashlib, algorithm),
    ).hexdigest()

    return hmac.compare_digest(received_hash, expected_hash)


@app.route("/webhooks/zenpays", methods=["POST"])
def handle_webhook():
    signature = request.headers.get("X-ZenPay-Signature", "")
    timestamp = request.headers.get("X-Zenpay-Timestamp", "")
    event_id = request.headers.get("X-Zenpay-Event-Id", "")

    # 1. Verify signature
    if not verify_webhook_signature(request.data, signature, WEBHOOK_SECRET):
        return jsonify({"error": "Invalid signature"}), 401

    # 2. Check timestamp freshness
    delivery_time = datetime.fromisoformat(timestamp.replace("Z", "+00:00"))
    now = datetime.now(timezone.utc)
    if abs((now - delivery_time).total_seconds()) > 300:
        return jsonify({"error": "Webhook timestamp expired"}), 401

    # 3. Parse and process
    event = request.get_json(force=True)
    handle_webhook_event(event)

    return "OK", 200
```

Replay Attack Prevention

Validate `X-Zenpay-Timestamp` is within 5 minutes of the current server time. This prevents attackers from replaying captured webhook deliveries.

Retry Behavior

If your endpoint fails to respond with a `2xx` status code within 30 seconds, ZenPay will retry the delivery:

- **Max retries**: 3 attempts
- **Backoff**: Exponential backoff between retries
- Each retry includes an incremented `X-Zenpay-Attempt` header

Best Practices

1. **Respond quickly** — Return a `2xx` response within 30 seconds. Queue events for asynchronous background processing.
2. **Handle duplicates** — Events may be delivered more than once. Use `X-Zenpay-Event-Id` to deduplicate.
3. **Use HTTPS** — Always use secure endpoints in production.
4. **Verify signatures** — Never trust unverified payloads. Always validate `X-ZenPay-Signature` using the raw request body.
5. **Check timestamps** — Validate `X-Zenpay-Timestamp` is within 5 minutes of current time to prevent replay attacks.
6. **Log everything** — Keep records of received webhooks for debugging.
7. **Monitor failures** — Check the dashboard for failed deliveries and fix endpoint issues promptly.

GUIDES

Testing

Learn how to test your ZenPays integration.

Test Environment

Use test API keys for development:

JAVASCRIPT

```javascript
const zenpays = new ZenPays({
  apiKey: process.env.ZENPAYS_TEST_API_KEY!, // zp_test_xxx
  baseUrl: 'https://api.zenpays.com',
})
```

PYTHON

```python
import os

zenpays = ZenPays(
    api_key=os.environ["ZENPAYS_TEST_API_KEY"],  # zp_test_xxx
    base_url="https://api.zenpays.com",
)
```

Test Payment Methods

Use these test values in the sandbox:

UPI

VPA	Scenario
`success@upi`	Successful payment
`failure@upi`	Payment failed
`timeout@upi`	Payment timeout

Net Banking

Bank Code	Scenario
`TEST_BANK_001`	Successful payment
`TEST_BANK_002`	Payment declined

Mocking the SDK

JAVASCRIPT

Using Vitest

```
import { describe, expect, it, vi } from 'vitest'
import { ZenPays } from '@zenxdigitalholdings/zenpays'

vi.mock('@zenxdigitalholdings/zenpays', () => ({
  ZenPays: vi.fn().mockImplementation(() => ({
    payments: {
      createPaymentIntent: vi.fn().mockResolvedValue({
        intentId: 'pi_test_xxx',
        status: 'pending',
      }),
    },
  })),
}))

describe('payment flow', () => {
  it('should create a payment intent', async () => {
    const zenpays = new ZenPays({ apiKey: 'test' })

    const intent = await zenpays.payments.createPaymentIntent({
      amount: 1000,
      currency: 'USD',
    })

    expect(intent.intentId).toBe('pi_test_xxx')
  })
})
```

Using Jest

```
import { ZenPays } from '@zenxdigitalholdings/zenpays'

jest.mock('@zenxdigitalholdings/zenpays')

const mockZenPays = ZenPays as jest.MockedClass<typeof ZenPays>

describe('payment flow', () => {
  beforeEach(() => {
    mockZenPays.mockImplementation(() => ({
      payments: {
        createPaymentIntent: jest.fn().mockResolvedValue({
          intentId: 'pi_test_xxx',
          status: 'pending',
        }),
      },
    } as any))
  })

  it('should create a payment intent', async () => {
    const zenpays = new ZenPays({ apiKey: 'test' })
    const intent = await zenpays.payments.createPaymentIntent({
      amount: 1000,
      currency: 'USD',
    })

    expect(intent.intentId).toBe('pi_test_xxx')
  })
})
```

PYTHON

Using unittest.mock

```
from unittest.mock import Mock, patch
import pytest
from zenpays import ZenPays


def test_create_payment_intent():
    with patch('zenpays.ZenPays') as MockZenPays:
        # Setup mock
        mock_instance = MockZenPays.return_value
        mock_instance.payments.create_payment_intent.return_value = {
            "intent_id": "pi_test_xxx",
            "status": "pending",
        }

        # Test code
        zenpays = ZenPays(api_key="test")
        intent = zenpays.payments.create_payment_intent({
            "amount": 1000,
            "currency": "USD",
        })

        assert intent["intent_id"] == "pi_test_xxx"
```

Using pytest-mock

```python
import pytest
from zenpays import ZenPays


def test_create_payment_intent(mocker):
    # Mock the ZenPays class
    mock_zenpays = mocker.Mock(spec=ZenPays)
    mock_zenpays.payments.create_payment_intent.return_value = {
        "intent_id": "pi_test_xxx",
        "status": "pending",
    }

    # Test code
    intent = mock_zenpays.payments.create_payment_intent({
        "amount": 1000,
        "currency": "USD",
    })

    assert intent["intent_id"] == "pi_test_xxx"
```

Integration Tests

Test against the sandbox API:

JAVASCRIPT

```javascript
import { ZenPays } from '@zenxdigitalholdings/zenpays'

describe('integration tests', () => {
  const zenpays = new ZenPays({
    apiKey: process.env.ZENPAYS_TEST_API_KEY!,
    baseUrl: 'https://api.zenpays.com',
  })

  it('should create and confirm a payment', async () => {
    // Create intent
    const intent = await zenpays.payments.createPaymentIntent({
      amount: 1000,
      currency: 'USD',
    })

    expect(intent.intentId).toBeDefined()

    // Confirm payment
    const result = await zenpays.payments.confirmPayment(intent.intentId, {
      customerDetails: {
        name: 'Test User',
        email: 'test@example.com',
        address: { country: 'US' },
      },
      paymentMethodDetails: {
        type: 'upi',
        vpa: 'test@upi',
      },
    })

    expect(result.status).toBe('succeeded')
  })
})
```

PYTHON

```python
import os
import pytest
from zenpays import ZenPays


@pytest.fixture
def zenpays():
    return ZenPays(
        api_key=os.environ["ZENPAYS_TEST_API_KEY"],
        base_url="https://api.zenpays.com",
    )


def test_create_and_confirm_payment(zenpays):
    # Create intent
    intent = zenpays.payments.create_payment_intent({
        "amount": 1000,
        "currency": "USD",
    })

    assert "intent_id" in intent

    # Confirm payment
    result = zenpays.payments.confirm_payment(intent["intent_id"], {
        "customer_details": {
            "name": "Test User",
            "email": "test@example.com",
            "address": {"country": "US"},
        },
        "payment_method_details": {
            "type": "upi",
            "vpa": "test@upi",
        },
    })

    assert result["status"] == "succeeded"
```

Custom Fetch for Testing

JAVASCRIPT

Provide a custom fetch implementation:

```
import { ZenPays } from '@zenxdigitalholdings/zenpays'

const mockFetch = vi.fn().mockResolvedValue({
  ok: true,
  json: () => Promise.resolve({
    success: true,
    data: { intentId: 'pi_xxx', status: 'pending' },
  }),
})

const zenpays = new ZenPays({
  apiKey: 'test',
  fetch: mockFetch,
})

// Now all requests go through mockFetch
await zenpays.payments.createPaymentIntent({ amount: 1000, currency: 'USD' })

expect(mockFetch).toHaveBeenCalledWith(
  expect.stringContaining('/payment-intent'),
  expect.objectContaining({ method: 'POST' })
)
```

PYTHON

Provide a custom session for testing:

```
from unittest.mock import Mock
from zenpays import ZenPays
import requests


def test_with_custom_session():
    # Create a mock session
    mock_session = Mock(spec=requests.Session)
    mock_response = Mock()
    mock_response.json.return_value = {
        "success": True,
        "data": {"intent_id": "pi_xxx", "status": "pending"},
    }
    mock_response.raise_for_status.return_value = None
    mock_session.post.return_value = mock_response

    # Create ZenPays with custom session
    zenpays = ZenPays(api_key="test", session=mock_session)

    # Now all requests go through mock_session
    intent = zenpays.payments.create_payment_intent({"amount": 1000, "currency": "USD"})

    # Verify the call
    mock_session.post.assert_called_once()
    assert "payment-intent" in mock_session.post.call_args[0][0]
```

Testing Webhooks

Use a tool like ngrok for local webhook testing:

```
ngrok http 3000
```

Then register the ngrok URL as your webhook endpoint:

JAVASCRIPT

```javascript
await zenpays.merchants.createWebhook({
  url: 'https://your-ngrok-url.ngrok.io/webhooks',
  events: ['payment.intent.succeeded'],
})
```

PYTHON

```python
zenpays.merchants.create_webhook({
    "url": "https://your-ngrok-url.ngrok.io/webhooks",
    "events": ["payment.intent.succeeded"],
})
```

GUIDES

TypeScript

The ZenPays SDK is written in TypeScript and provides full type safety.

Type Imports

Import types alongside runtime exports:

```
import type { Customer, PaymentIntent, Transaction } from '@zenxdigitalholdings/zenpays'
import {

  ZenPays
} from '@zenxdigitalholdings/zenpays'
```

Or import types from the types subpath:

```
import type {
  CreatePaymentIntentRequest,
  Customer,
  PaymentIntent,
} from '@zenxdigitalholdings/zenpays/types'
```

Type-Safe API Calls

All API methods are fully typed:

```
const zenpays = new ZenPays({ apiKey: 'xxx' })

// Request type is inferred
const intent = await zenpays.payments.createPaymentIntent({
  amount: 1000,
  currency: 'USD',
  // TypeScript will error if you miss required fields
})

// Response type is PaymentIntentResponse
console.log(intent.intentId) // string
console.log(intent.status) // PaymentStatus
```

Generic Response Types

Paginated responses use generics:

```
import type { PaginatedResponse, Transaction } from '@zenxdigitalholdings/zenpays'

const result: PaginatedResponse<Transaction>
  = await zenpays.payments.listTransactions()

result.data // Transaction[]
result.total // number
result.page // number
```

Discriminated Unions

Status types use discriminated unions for type narrowing:

```
const result = await zenpays.payments.confirmPayment('pi_xxx', { ... })

if (result.nextAction?.type === 'redirect') {
  // TypeScript knows redirectUrl exists
  window.location.href = result.nextAction.redirectUrl!
} else if (result.nextAction?.type === 'three_ds') {
  // TypeScript knows threeDsUrl exists
  handle3DSecure(result.nextAction.threeDsUrl!)
}
```

Error Types

Error classes are also typed:

```
import {
  ZenPaysError,
  ValidationError,
  RateLimitError,
} from '@zenxdigitalholdings/zenpays'

try {
  await zenpays.payments.createPaymentIntent({ ... })
} catch (error) {
  if (error instanceof ValidationError) {
    // TypeScript knows about .fields
    console.log(error.fields)
  } else if (error instanceof RateLimitError) {
    // TypeScript knows about .retryAfter
    console.log(error.retryAfter)
  }
}
```

Utility Types

The SDK exports utility types:

```
import type {
  Currency,
  PaymentMethodType,
  PaymentStatus,
  RiskLevel,
} from '@zenxdigitalholdings/zenpays'

// Use in your own types
interface Order {
  id: string
  paymentStatus: PaymentStatus
  paymentMethod: PaymentMethodType
  currency: Currency
}
```

Configuration Types

```
import type { ZenPaysConfig } from '@zenxdigitalholdings/zenpays'

// Type-safe configuration
const config: ZenPaysConfig = {
  apiKey: process.env.ZENPAYS_API_KEY!,
  timeout: 30000,
}
```

Strict Mode

The SDK works best with strict TypeScript settings:

```
{
  "compilerOptions": {
    "strict": true,
    "strictNullChecks": true
  }
}
```

PART

Examples

6 documents

EXAMPLES

Basic Payment

A complete example of creating and processing a payment.

JAVASCRIPT

```
import { PaymentError, ValidationError, ZenPays } from '@zenxdigitalholdings/zenpays'

async function processPayment() {
  const zenpays = new ZenPays({
    apiKey: process.env.ZENPAYS_API_KEY!,
  })

  try {
    // 1. Create a payment intent
    const intent = await zenpays.payments.createPaymentIntent({
      amount: 2999,
      currency: 'INR',
      paymentMethod: 'upi',
      customerEmail: 'john@example.com',
      customerFirstName: 'John',
      customerLastName: 'Doe',
      customerCountry: 'IN',
      description: 'Pro Plan - Monthly',
    })

    console.log('Payment Intent created:', intent.intentId)

    // 2. Confirm with payment details
    const result = await zenpays.payments.confirmPayment(intent.intentId, {
      customerDetails: {
        name: 'John Doe',
        email: 'john@example.com',
        address: { country: 'US' },
      },
      paymentMethodDetails: {
        type: 'upi',
        vpa: 'john@upi',
      },
    })

    // 3. Handle the result
    switch (result.status) {
      case 'succeeded':
        console.log('Payment successful!')
        console.log('Transaction ID:', result.externalTransactionId)
        return { success: true, transactionId: result.externalTransactionId }

      case 'processing':
        console.log('Payment is processing...')
        return { success: false, status: 'processing' }

      case 'failed':
        console.log('Payment failed:', result.message)
        return { success: false, error: result.message }

      default:
        // Handle next actions (3D Secure, etc.)
        if (result.nextAction) {
          return handleNextAction(result.nextAction)
        }
    }
  }
  catch (error) {
    if (error instanceof ValidationError) {
      console.error('Validation error:', error.message)
      return { success: false, error: 'Invalid payment details' }
    }
```

```
    if (error instanceof PaymentError) {
      console.error('Payment error:', error.message)
      return { success: false, error: 'Payment declined' }
    }

    throw error
  }
}

function handleNextAction(nextAction: any) {
  if (nextAction.type === 'redirect') {
    console.log('Redirect required:', nextAction.redirectUrl)
    return { success: false, redirect: nextAction.redirectUrl }
  }

  if (nextAction.type === 'three_ds') {
    console.log('3D Secure required')
    return { success: false, threeDsUrl: nextAction.threeDsUrl }
  }

  return { success: false, error: 'Unknown action required' }
}

// Run the example
processPayment()
  .then(result => console.log('Result:', result))
  .catch(console.error)
```

PYTHON

```python
import os
from zenpays import ZenPays
from zenpays.errors import PaymentError, ValidationError


def process_payment():
    zenpays = ZenPays(api_key=os.environ["ZENPAYS_API_KEY"])

    try:
        # 1. Create a payment intent
        intent = zenpays.payments.create_payment_intent({
            "amount": 2999,
            "currency": "INR",
            "paymentMethod": "upi",
            "customerEmail": "john@example.com",
            "customerFirstName": "John",
            "customerLastName": "Doe",
            "customerCountry": "IN",
            "description": "Pro Plan - Monthly",
        })

        print(f"Payment Intent created: {intent['intent_id']}")

        # 2. Confirm with payment details
        result = zenpays.payments.confirm_payment(intent["intent_id"], {
            "customer_details": {
                "name": "John Doe",
                "email": "john@example.com",
                "address": {"country": "US"},
            },
            "payment_method_details": {
                "type": "upi",
                "vpa": "john@upi",
            },
        })

        # 3. Handle the result
        status = result["status"]

        if status == "succeeded":
            print("Payment successful!")
            print(f"Transaction ID: {result['external_transaction_id']}")
            return {"success": True, "transaction_id": result["external_transaction_id"]}

        elif status == "processing":
            print("Payment is processing...")
            return {"success": False, "status": "processing"}

        elif status == "failed":
            print(f"Payment failed: {result['message']}")
            return {"success": False, "error": result["message"]}

        else:
            # Handle next actions (3D Secure, etc.)
            if "next_action" in result:
                return handle_next_action(result["next_action"])

    except ValidationError as e:
        print(f"Validation error: {e}")
        return {"success": False, "error": "Invalid payment details"}

    except PaymentError as e:
```

```python
        print(f"Payment error: {e}")
        return {"success": False, "error": "Payment declined"}


def handle_next_action(next_action):
    if next_action["type"] == "redirect":
        print(f"Redirect required: {next_action['redirect_url']}")
        return {"success": False, "redirect": next_action["redirect_url"]}

    if next_action["type"] == "three_ds":
        print("3D Secure required")
        return {"success": False, "three_ds_url": next_action["three_ds_url"]}

    return {"success": False, "error": "Unknown action required"}


# Run the example
if __name__ == "__main__":
    result = process_payment()
    print(f"Result: {result}")
```

EXAMPLES

Checkout Flow

Create a checkout session for on-ramp payments.

JAVASCRIPT

```javascript
import { ZenPays } from '@zenxdigitalholdings/zenpays'

async function createCheckout() {
  const zenpays = new ZenPays({
    apiKey: process.env.ZENPAYS_API_KEY!,
  })

  // 1. Detect customer's location
  const geo = await zenpays.checkout.detectGeo()
  console.log('Detected country:', geo.country, 'Currency:', geo.currency)

  // 2. Create checkout session
  const session = await zenpays.checkout.createSession({
    playerId: 'player_123',
    amount: 100, // Amount in fiat currency
    targetSettlementAsset: 'USDT',
    successUrl: 'https://example.com/success',
    cancelUrl: 'https://example.com/cancel',
  })

  console.log('Session created:', session.id)

  // 3. Get available banks (if bank transfer)
  const { banks } = await zenpays.checkout.getBanks(geo.currency)
  console.log('Available banks:', banks.length)

  // 4. Select optimal provider
  const provider = await zenpays.checkout.selectProvider({
    sessionId: session.id,
    playerGeo: geo.country,
    currency: geo.currency,
    amount: 100,
    paymentMethod: 'card',
    targetSettlementAsset: 'USDT',
  })

  console.log('Selected provider:', provider.provider)
  console.log('Widget URL:', provider.widgetUrl)
  console.log('Embed mode:', provider.embedMode)

  // 5. Redirect or embed widget
  if (provider.embedMode === 'redirect') {
    // Redirect to provider
    window.location.href = provider.widgetUrl
  }
  else {
    // Embed in iframe
    const iframe = document.createElement('iframe')
    iframe.src = provider.widgetUrl
    iframe.style.width = '100%'
    iframe.style.height = '600px'
    document.body.appendChild(iframe)
  }

  // 6. Monitor order status
  const checkStatus = async () => {
    const order = await zenpays.checkout.getRampOrder(session.rampOrderId!)

    switch (order.status) {
      case 'pending':
        console.log('Waiting for payment...')
        break
```

```
      case 'pending_funds':
        console.log('Payment received, waiting for confirmation...')
        break
      case 'confirmed':
        console.log('Payment confirmed!')
        break
      case 'matched':
        console.log('Crypto matched!')
        break
      case 'cleared':
        console.log('Order complete!')
        return true
      case 'failed':
        console.log('Order failed')
        return false
    }

    // Check again in 5 seconds
    setTimeout(checkStatus, 5000)
  }

  checkStatus()
}

createCheckout()
```

PYTHON

```python
import os
import time
from zenpays import ZenPays


def create_checkout():
    zenpays = ZenPays(api_key=os.environ["ZENPAYS_API_KEY"])

    # 1. Detect customer's location
    geo = zenpays.checkout.detect_geo()
    print(f"Detected country: {geo['country']}, Currency: {geo['currency']}")

    # 2. Create checkout session
    session = zenpays.checkout.create_session({
        "player_id": "player_123",
        "amount": 100,  # Amount in fiat currency
        "target_settlement_asset": "USDT",
        "success_url": "https://example.com/success",
        "cancel_url": "https://example.com/cancel",
    })

    print(f"Session created: {session['id']}")

    # 3. Get available banks (if bank transfer)
    result = zenpays.checkout.get_banks(geo["currency"])
    banks = result["banks"]
    print(f"Available banks: {len(banks)}")

    # 4. Select optimal provider
    provider = zenpays.checkout.select_provider({
        "session_id": session["id"],
        "player_geo": geo["country"],
        "currency": geo["currency"],
        "amount": 100,
        "payment_method": "card",
        "target_settlement_asset": "USDT",
    })

    print(f"Selected provider: {provider['provider']}")
    print(f"Widget URL: {provider['widget_url']}")
    print(f"Embed mode: {provider['embed_mode']}")

    # 5. Redirect or embed widget
    if provider["embed_mode"] == "redirect":
        # Redirect to provider
        print(f"Redirect to: {provider['widget_url']}")
        # In a web app: redirect user to provider["widget_url"]
    else:
        # Embed in iframe
        print("Embed widget in iframe")
        # In a web app: create iframe with provider["widget_url"]

    # 6. Monitor order status
    def check_status():
        order = zenpays.checkout.get_ramp_order(session["ramp_order_id"])

        status = order["status"]

        if status == "pending":
            print("Waiting for payment...")
        elif status == "pending_funds":
            print("Payment received, waiting for confirmation...")
```

```
        elif status == "confirmed":
            print("Payment confirmed!")
        elif status == "matched":
            print("Crypto matched!")
        elif status == "cleared":
            print("Order complete!")
            return True
        elif status == "failed":
            print("Order failed")
            return False

        # Check again in 5 seconds
        time.sleep(5)
        return check_status()

    check_status()


if __name__ == "__main__":
    create_checkout()
```

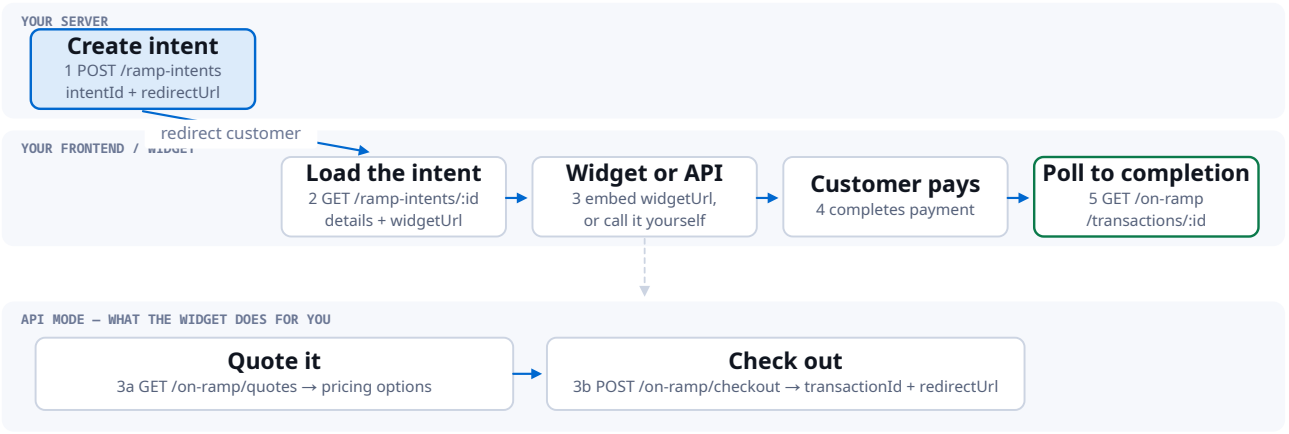
EXAMPLES

On-Ramp (Buy) Flow

This guide walks through the complete on-ramp integration — allowing your customers to buy crypto with fiat. The flow uses **ramp intents** to track the entire lifecycle from creation to completion.

Flow Overview

You create the intent server-side, then hand the customer to the widget or drive the API yourself.



Poll until the transaction reports completed — or subscribe to the ramp webhooks instead of polling.

Buying crypto: intent on your server, purchase in your frontend

Step 1: Create a Ramp Intent (Server-Side)

Create a ramp intent from your backend. This requires your API key and merchant ID.

JAVASCRIPT

```javascript
// Server-side — requires API key
const createBuyIntent = async (customerEmail, walletAddress) => {
  const response = await fetch('https://api.zenpays.com/payment/api/v1/ramp-intents', {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json',
      'Authorization': `Bearer ${process.env.ZENPAYS_API_KEY}`,
      'x-merchant-id': process.env.ZENPAYS_MERCHANT_ID,
      'x-request-id': `req_${Date.now()}`,
    },
    body: JSON.stringify({
      type: 'buy',
      wallets: [{ network: 'ethereum', address: walletAddress }],
      fiatCurrency: 'USD',
      cryptoCurrency: 'eth_ethereum',
      amount: 100,
      customerEmail,
      successUrl: 'https://yourapp.com/buy/success',
      cancelUrl: 'https://yourapp.com/buy/cancel',
    }),
  });

  const { data } = await response.json();
  // data = { intentId, redirectUrl, expiresAt }
  return data;
};
```

PYTHON

```python
import requests
import os
import time

def create_buy_intent(customer_email: str, wallet_address: str):
    response = requests.post(
        "https://api.zenpays.com/payment/api/v1/ramp-intents",
        headers={
            "Authorization": f"Bearer {os.environ['ZENPAYS_API_KEY']}",
            "x-merchant-id": os.environ["ZENPAYS_MERCHANT_ID"],
            "x-request-id": f"req_{int(time.time())}",
        },
        json={
            "type": "buy",
            "wallets": [{"network": "ethereum", "address": wallet_address}],
            "fiatCurrency": "USD",
            "cryptoCurrency": "eth_ethereum",
            "amount": 100,
            "customerEmail": customer_email,
            "successUrl": "https://yourapp.com/buy/success",
            "cancelUrl": "https://yourapp.com/buy/cancel",
        },
    )
    return response.json()["data"]
```

Step 2: Redirect Customer to Widget

Send the customer to the `redirectUrl` from Step 1, or fetch the intent on your frontend to get the widget URL for iframe embedding.

JAVASCRIPT

```javascript
// Option A: Simple redirect
window.location.href = intentData.redirectUrl;

// Option B: Fetch intent and embed widget in iframe
const fetchAndEmbed = async (intentId) => {
  const response = await fetch(
    `https://api.zenpays.com/payment/api/v1/ramp-intents/${intentId}`
  );
  const { data } = await response.json();

  // Terminal lifecycle states
  if (data.intent.status === 'expired') {
    showError('This session has expired. Please start again.');
    return;
  }
  if (data.intent.status === 'cancelled') {
    // When KYC is declined the intent transitions to cancelled with
    // cancelReason='kyc_rejected'. See get-ramp-intent for details.
    showError(
      data.intent.cancelReason === 'kyc_rejected'
        ? 'Identity verification could not be completed.'
        : 'This session has been cancelled.'
    );
    return;
  }

  // KYC gate — surface a friendly state-specific message
  switch (data.intent.kycStatus) {
    case 'in_review':
      showInfo('Verification under review — we\'ll continue automatically once approved.');
      return;
    case 'pending':
      // Customer needs to start / finish verification. If `kycRequired` is
      // present in the response and `kycVerificationUrl` is set, embed it;
      // otherwise call initiate-kyc to create a session.
      break;
  }

  if (data.widgetUrl) {
    const iframe = document.createElement('iframe');
    iframe.src = data.widgetUrl;
    iframe.style.cssText = 'width:100%;height:600px;border:none;border-radius:12px;';
    iframe.allow = 'payment';
    document.getElementById('widget-container').appendChild(iframe);
  }
};
```

PYTHON

```python
# Server-side: return the redirect URL to your frontend
def get_intent_widget(intent_id: str):
    response = requests.get(
        f"https://api.zenpays.com/payment/api/v1/ramp-intents/{intent_id}"
    )
    data = response.json()["data"]

    if data["intent"]["status"] == "expired":
        raise Exception("Intent expired")

    return {
        "intent": data["intent"],
        "widgetUrl": data["widgetUrl"],
    }
```

Step 3: Get Quotes (API Mode)

If you're building a custom UI instead of using the widget, fetch buy quotes to show pricing options.

JAVASCRIPT

```javascript
const getQuotes = async (source, destination, amount) => {
  const params = new URLSearchParams({ source, destination, amount: String(amount) });

  const response = await fetch(
    `https://api.zenpays.com/payment/api/v1/on-ramp/quotes?${params}`
  );
  const { data } = await response.json();

  // data.quotes = array of quote options
  data.quotes.forEach((quote) => {
    console.log(`${quote.onramp}: ${quote.outputAmount} crypto`);
    console.log(`  Rate: ${quote.rate}, Fee: ${quote.totalFee}`);
    console.log(`  Payment: ${quote.paymentMethod}`);
    if (quote.recommended) console.log('  ☆ Recommended');
  });

  return data.quotes;
};

const quotes = await getQuotes('usd', 'eth_ethereum', 100);
```

PYTHON

```python
def get_quotes(source: str, destination: str, amount: float):
    response = requests.get(
        "https://api.zenpays.com/payment/api/v1/on-ramp/quotes",
        params={
            "source": source,
            "destination": destination,
            "amount": amount,
        },
    )
    quotes = response.json()["data"]["quotes"]

    for quote in quotes:
        print(f"{quote['onramp']}: {quote['outputAmount']} crypto")
        print(f"  Rate: {quote['rate']}, Fee: {quote['totalFee']}")
    return quotes
```

Step 4: Create Checkout

After the customer selects a quote, create a checkout to start the payment.

JAVASCRIPT

```javascript
const createCheckout = async (selectedQuote, walletAddress) => {
  const response = await fetch(
    'https://api.zenpays.com/payment/api/v1/on-ramp/checkout',
    {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({
        onramp: selectedQuote.onramp,
        source: 'usd',
        destination: 'eth_ethereum',
        amount: 100,
        type: 'buy',
        paymentMethod: selectedQuote.paymentMethod,
        walletAddress,
      }),
    }
  );

  const { data } = await response.json();
  // Redirect to payment page
  window.location.href = data.redirectUrl;
};
```

PYTHON

```python
def create_checkout(selected_quote, wallet_address: str):
    response = requests.post(
        "https://api.zenpays.com/payment/api/v1/on-ramp/checkout",
        json={
            "onramp": selected_quote["onramp"],
            "source": "usd",
            "destination": "eth_ethereum",
            "amount": 100,
            "type": "buy",
            "paymentMethod": selected_quote["paymentMethod"],
            "walletAddress": wallet_address,
        },
    )
    return response.json()["data"]
```

Step 5: Poll Transaction Status

After the customer completes payment, poll the transaction status until it reaches a terminal state.

JAVASCRIPT

```javascript
const pollTransaction = async (transactionId) => {
  const terminalStatuses = ['completed', 'failed', 'expired', 'refunded'];

  while (true) {
    const response = await fetch(
      `https://api.zenpays.com/payment/api/v1/on-ramp/transactions/${transactionId}`
    );
    const { data } = await response.json();

    console.log(`Status: ${data.status}`);

    if (terminalStatuses.includes(data.status)) {
      return data;
    }

    // Wait 5 seconds before next poll
    await new Promise((resolve) => setTimeout(resolve, 5000));
  }
};

const result = await pollTransaction('txn_abc123');

if (result.status === 'completed') {
  console.log(`Success! ${result.outAmount} crypto delivered`);
  console.log(`TX Hash: ${result.transactionHash}`);
} else {
  console.log(`Transaction ${result.status}`);
}
```

PYTHON

```python
import time

def poll_transaction(transaction_id: str):
    terminal = {"completed", "failed", "expired", "refunded"}

    while True:
        response = requests.get(
            f"https://api.zenpays.com/payment/api/v1/on-ramp/transactions/{transaction_id}"
        )
        data = response.json()["data"]
        print(f"Status: {data['status']}")

        if data["status"] in terminal:
            return data

        time.sleep(5)

result = poll_transaction("txn_abc123")
if result["status"] == "completed":
    print(f"Success! {result['outAmount']} crypto delivered")
```

Step 6: Receive Webhook Notification

Instead of polling (or in addition to it), you can receive a webhook when the ramp completes or fails. Configure your webhook URL in the merchant dashboard.

`ramp.completed` — crypto has been sent to the customer's wallet:

```json
{
  "event_type": "ramp.completed",
  "payment_data": {
    "merchant_id": "m_abc123",
    "intentId": "ri_1710345678000_a1b2c3d4e5f6g7h8",
    "type": "buy",
    "status": "completed",
    "fiatCurrency": "USD",
    "cryptoCurrency": "eth_ethereum",
    "amount": 100,
    "chain": "ethereum",
    "destinationAddress": "0x742d35Cc6634C0532925a3b844Bc9e7595f2bD28",
    "fireblocksTxId": "fb_tx_9a8b7c6d5e4f",
    "completedAt": "2026-03-16T14:30:00.000Z"
  }
}
```

`ramp.failed` — crypto send failed:

```
{
  "event_type": "ramp.failed",
  "payment_data": {
    "merchant_id": "m_abc123",
    "intentId": "ri_1710345678000_a1b2c3d4e5f6g7h8",
    "type": "buy",
    "status": "failed",
    "fiatCurrency": "USD",
    "cryptoCurrency": "eth_ethereum",
    "amount": 100,
    "reason": "Crypto send failed",
    "failedAt": "2026-03-16T14:30:00.000Z"
  }
}
```

Webhook vs Polling

We recommend using webhooks as the primary notification mechanism and polling as a fallback. Webhooks are delivered as soon as the status changes — no delay. See the Webhooks guide for setup instructions, signature verification, and retry behavior.

Intent Status Lifecycle

```
created  →  active  →  completed
                    →  cancelled
                    →  expired (auto, after 1 hour)
```

Status	Description
`created`	Intent created, not yet accessed
`active`	Customer has accessed the intent (first GET)
`completed`	Transaction completed successfully
`cancelled`	Manually cancelled by merchant or customer
`expired`	Auto-expired after 1 hour

Error Handling

- **Intent expired**: Create a new intent and redirect the customer
- **Quote expired**: Fetch fresh quotes — prices change frequently
- **Payment failed**: Check the transaction status for details; the customer can retry
- **Network errors**: Implement retry with exponential backoff

Next Steps

- Create Ramp Intent (https://docs.zenpays.com/docs/rest-api/endpoints/ramp/create-ramp-intent) — API reference
- Buy Quotes (https://docs.zenpays.com/docs/rest-api/endpoints/ramp/buy-quotes) — Quote endpoint details
- Buy Checkout (https://docs.zenpays.com/docs/rest-api/endpoints/ramp/buy-checkout) — Checkout endpoint details
- Transaction Status (https://docs.zenpays.com/docs/rest-api/endpoints/ramp/transaction-status) — Status polling details
- Off-Ramp Flow — Sell crypto integration guide

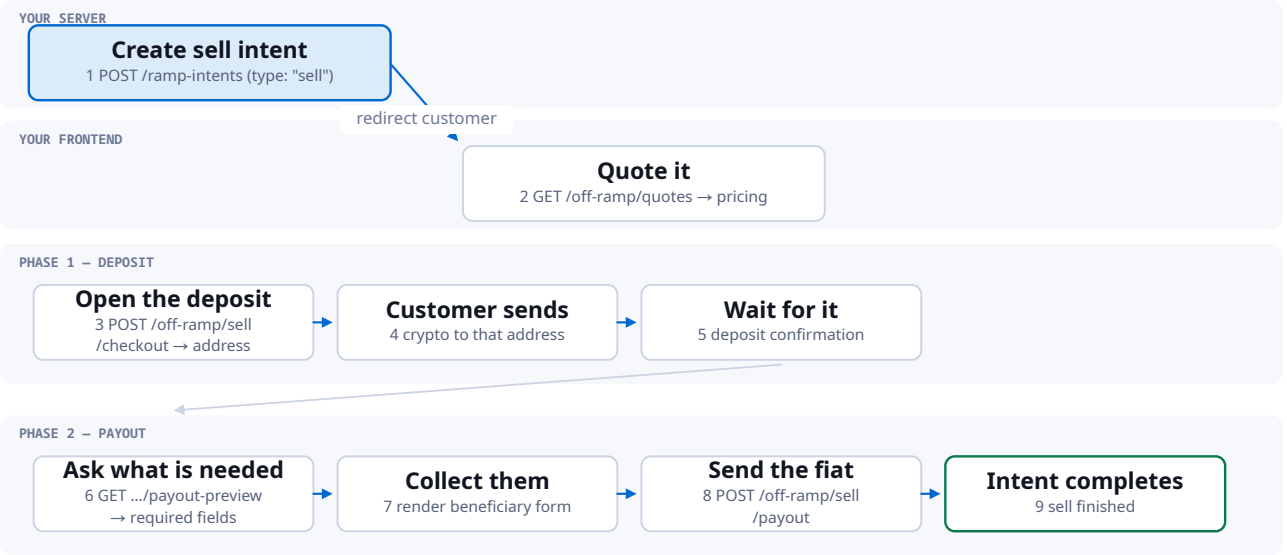
EXAMPLES

Off-Ramp (Sell) Flow

This guide walks through the complete off-ramp integration — allowing your customers to sell crypto and receive fiat. The sell flow uses a **two-phase process**: Phase 1 generates a crypto deposit address, and Phase 2 submits beneficiary details for the fiat payout.

Flow Overview

Selling runs in two phases. The customer deposits crypto, and only then do you collect beneficiary details and pay out.



The required beneficiary fields differ by corridor, which is why step 6 exists — do not hardcode the form.

Selling crypto: deposit first, then the fiat payout

Step 1: Create a Sell Intent (Server-Side)

Create a ramp intent with `type: "sell"` from your backend.

JAVASCRIPT

```javascript
// Server-side — requires API key
const createSellIntent = async (customerEmail, walletAddress) => {
  const response = await fetch('https://api.zenpays.com/payment/api/v1/ramp-intents', {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json',
      'Authorization': `Bearer ${process.env.ZENPAYS_API_KEY}`,
      'x-merchant-id': process.env.ZENPAYS_MERCHANT_ID,
      'x-request-id': `req_${Date.now()}`,
    },
    body: JSON.stringify({
      type: 'sell',
      wallets: [{ network: 'tron', address: walletAddress }],
      fiatCurrency: 'USD',
      cryptoCurrency: 'usdt_tron',
      customerEmail,
      successUrl: 'https://yourapp.com/sell/success',
      cancelUrl: 'https://yourapp.com/sell/cancel',
    }),
  });

  const { data } = await response.json();
  return data; // { intentId, redirectUrl, expiresAt }
};
```

PYTHON

```python
import requests
import os
import time

def create_sell_intent(customer_email: str, wallet_address: str):
    response = requests.post(
        "https://api.zenpays.com/payment/api/v1/ramp-intents",
        headers={
            "Authorization": f"Bearer {os.environ['ZENPAYS_API_KEY']}",
            "x-merchant-id": os.environ["ZENPAYS_MERCHANT_ID"],
            "x-request-id": f"req_{int(time.time())}",
        },
        json={
            "type": "sell",
            "wallets": [{"network": "tron", "address": wallet_address}],
            "fiatCurrency": "USD",
            "cryptoCurrency": "usdt_tron",
            "customerEmail": customer_email,
            "successUrl": "https://yourapp.com/sell/success",
            "cancelUrl": "https://yourapp.com/sell/cancel",
        },
    )
    return response.json()["data"]
```

Step 2: Get Sell Quotes

Fetch quotes to show the customer how much fiat they'll receive for their crypto.

JAVASCRIPT

```javascript
const getSellQuotes = async (cryptoAmount) => {
  const params = new URLSearchParams({
    source: 'usdt_tron',
    destination: 'USD',
    amount: String(cryptoAmount),
  });

  const response = await fetch(
    `https://api.zenpays.com/payment/api/v1/off-ramp/quotes?${params}`
  );
  const { data } = await response.json();

  data.quotes.forEach((quote) => {
    console.log(`Provider: ${quote.onramp}`);
    console.log(`  You send: ${quote.inputAmount} USDT`);
    console.log(`  You receive: ${quote.outputAmount} USD`);
    console.log(`  Rate: ${quote.rate}, Fee: ${quote.totalFee}`);
  });

  return data.quotes;
};

const quotes = await getSellQuotes(20);
const selectedQuote = quotes[0]; // Or let user choose
```

PYTHON

```python
def get_sell_quotes(crypto_amount: float):
    response = requests.get(
        "https://api.zenpays.com/payment/api/v1/off-ramp/quotes",
        params={
            "source": "usdt_tron",
            "destination": "USD",
            "amount": crypto_amount,
        },
    )
    quotes = response.json()["data"]["quotes"]

    for quote in quotes:
        print(f"Provider: {quote['onramp']}")
        print(f"  Send: {quote['inputAmount']} USDT → Receive: {quote['outputAmount']} USD")
        print(f"  Rate: {quote['rate']}, Fee: {quote['totalFee']}")
    return quotes
```

Step 3: Phase 1 — Create Sell Deposit

Generate a deposit address for the customer to send crypto to.

JAVASCRIPT

```javascript
const createDeposit = async (intentId, quote) => {
  const response = await fetch(
    'https://api.zenpays.com/payment/api/v1/off-ramp/sell/checkout',
    {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({
        intentId,
        cryptoCurrency: 'USDT',
        chain: 'tron',
        fiatCurrency: 'USD',
        cryptoAmount: quote.inputAmount,
        fiatAmount: quote.outputAmount,
        rate: quote.rate,
        totalFee: quote.totalFee,
      }),
    }
  );

  const { data } = await response.json();

  // Display to customer
  console.log(`Send ${data.expectedCryptoAmount} ${data.currency} to:`);
  console.log(`Address: ${data.depositAddress}`);
  console.log(`Network: ${data.chain}`);
  if (data.memo) console.log(`Memo: ${data.memo}`);

  // Save for Phase 2
  return {
    cryptoDepositId: data.cryptoDepositId,
    depositAddress: data.depositAddress,
    expectedAmount: data.expectedCryptoAmount,
  };
};
```

PYTHON

```python
def create_deposit(intent_id: str, quote: dict):
    response = requests.post(
        "https://api.zenpays.com/payment/api/v1/off-ramp/sell/checkout",
        json={
            "intentId": intent_id,
            "cryptoCurrency": "USDT",
            "chain": "tron",
            "fiatCurrency": "USD",
            "cryptoAmount": quote["inputAmount"],
            "fiatAmount": quote["outputAmount"],
            "rate": quote["rate"],
            "totalFee": quote["totalFee"],
        },
    )
    data = response.json()["data"]

    print(f"Send {data['expectedCryptoAmount']} {data['currency']} to:")
    print(f"Address: {data['depositAddress']}")
    print(f"Network: {data['chain']}")

    return {
        "cryptoDepositId": data["cryptoDepositId"],
        "depositAddress": data["depositAddress"],
    }
```

Display to Customer

Show the deposit address with a QR code and a countdown timer based on `expiresAt`. Remind the customer to send the exact `expectedCryptoAmount` on the correct chain.

Step 4: Wait for Deposit Confirmation

After the customer sends crypto, the system automatically detects and confirms the deposit. You can poll the intent status or listen for WebSocket events.

JAVASCRIPT

```javascript
const waitForDeposit = async (intentId) => {
  console.log('Waiting for crypto deposit...');

  while (true) {
    const response = await fetch(
      `https://api.zenpays.com/payment/api/v1/ramp-intents/${intentId}`
    );
    const { data } = await response.json();
    const phase = data.intent.metadata?.phase;

    if (phase === 'awaiting_payout' || phase === 'completed') {
      console.log('Deposit confirmed!');
      return data.intent;
    }

    if (data.intent.status === 'expired' || data.intent.status === 'cancelled') {
      throw new Error(`Intent ${data.intent.status}`);
    }

    // Poll every 10 seconds
    await new Promise((resolve) => setTimeout(resolve, 10000));
  }
};
```

PYTHON

```python
def wait_for_deposit(intent_id: str):
    print("Waiting for crypto deposit...")

    while True:
        response = requests.get(
            f"https://api.zenpays.com/payment/api/v1/ramp-intents/{intent_id}"
        )
        data = response.json()["data"]
        phase = (data["intent"].get("metadata") or {}).get("phase")

        if phase in ("awaiting_payout", "completed"):
            print("Deposit confirmed!")
            return data["intent"]

        if data["intent"]["status"] in ("expired", "cancelled"):
            raise Exception(f"Intent {data['intent']['status']}")

        time.sleep(10)
```

Step 5: Phase 2a — Get Payout Preview

Once the deposit is confirmed, fetch the required beneficiary fields for the fiat payout.

JAVASCRIPT

```javascript
const getPayoutPreview = async (intentId, fiatCurrency, country = 'US') => {
  const params = new URLSearchParams({ intentId, fiatCurrency, country });

  const response = await fetch(
    `https://api.zenpays.com/payment/api/v1/off-ramp/sell/payout-preview?${params}`
  );
  const { data } = await response.json();

  console.log(`Provider: ${data.tspDisplayName}`);
  console.log(`Payout: ${data.fiatAmount} ${data.fiatCurrency}`);
  console.log(`Fees: ${data.fees.total}`);
  console.log(`Required fields: ${data.requiredFields.length}`);

  return data;
};

const preview = await getPayoutPreview(intentId, 'USD', 'US');

// Dynamically render form from requiredFields
preview.requiredFields.forEach((field) => {
  // Create form inputs based on field.type, field.label, field.required, etc.
  console.log(`  ${field.required ? '*' : ' '} ${field.label} (${field.fieldName})`);
});
```

PYTHON

```python
def get_payout_preview(intent_id: str, fiat_currency: str, country: str = "US"):
    response = requests.get(
        "https://api.zenpays.com/payment/api/v1/off-ramp/sell/payout-preview",
        params={
            "intentId": intent_id,
            "fiatCurrency": fiat_currency,
            "country": country,
        },
    )
    data = response.json()["data"]

    print(f"Provider: {data['tspDisplayName']}")
    print(f"Payout: {data['fiatAmount']} {data['fiatCurrency']}")
    print(f"Fees: {data['fees']['total']}")
    print(f"\nRequired fields:")
    for field in data["requiredFields"]:
        print(f"  {'*' if field['required'] else ' '} {field['label']} ({field['fieldName']})")
    return data
```

Step 6: Phase 2b — Submit Payout

Collect the beneficiary details from the customer and submit the payout.

JAVASCRIPT

```javascript
const submitPayout = async (intentId, cryptoDepositId, beneficiaryDetails) => {
  const response = await fetch(
    'https://api.zenpays.com/payment/api/v1/off-ramp/sell/payout',
    {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({
        intentId,
        cryptoDepositId,
        fiatCurrency: 'USD',
        country: 'US',
        beneficiaryDetails,
      }),
    }
  );

  const { data } = await response.json();

  console.log(`Payout ID: ${data.payoutId}`);
  console.log(`Status: ${data.status}`);
  console.log(`Amount: ${data.amount} ${data.currency}`);

  return data;
};

// Example: customer filled in the form
const payout = await submitPayout(intentId, deposit.cryptoDepositId, {
  receiverFirstName: 'John',
  receiverLastName: 'Doe',
  receiverCountry: 'US',
  receiverAccountNumber: '123456789',
  receiverBankName: 'Bank of America',
  receiverBankCode: 'BOFAUS3N',
  receiverAddressLine1: '123 Main St',
  receiverCity: 'New York',
  receiverState: 'NY',
  receiverPinCode: '10001',
  remittancePurpose: 'PAYP001 - Family Support',
  sourceOfFund: 'PAYF001 - Salary',
  relationship: 'PAYR001 - Self',
});

// Intent is now completed — redirect customer
window.location.href = '/sell/success';
```

PYTHON

```python
def submit_payout(intent_id: str, crypto_deposit_id: str, beneficiary_details: dict):
    response = requests.post(
        "https://api.zenpays.com/payment/api/v1/off-ramp/sell/payout",
        json={
            "intentId": intent_id,
            "cryptoDepositId": crypto_deposit_id,
            "fiatCurrency": "USD",
            "country": "US",
            "beneficiaryDetails": beneficiary_details,
        },
    )
    data = response.json()["data"]
    print(f"Payout ID: {data['payoutId']}")
    print(f"Status: {data['status']}")
    print(f"Amount: {data['amount']} {data['currency']}")
    return data

# Example
payout = submit_payout(intent_id, deposit["cryptoDepositId"], {
    "receiverFirstName": "John",
    "receiverLastName": "Doe",
    "receiverCountry": "US",
    "receiverAccountNumber": "123456789",
    "receiverBankName": "Bank of America",
    "receiverBankCode": "BOFAUS3N",
    "receiverAddressLine1": "123 Main St",
    "receiverCity": "New York",
    "receiverState": "NY",
    "receiverPinCode": "10001",
    "remittancePurpose": "PAYP001 - Family Support",
    "sourceOfFund": "PAYF001 - Salary",
    "relationship": "PAYR001 - Self",
})
```

Step 7: Receive Webhook Notification

After the payout is submitted (or if it fails), ZenPay delivers a webhook to your configured endpoint.

`ramp.completed` — fiat payout has been submitted successfully:

```json
{
  "event_type": "ramp.completed",
  "payment_data": {
    "merchant_id": "m_abc123",
    "intentId": "ri_1710345678000_a1b2c3d4e5f6g7h8",
    "type": "sell",
    "status": "completed",
    "fiatCurrency": "USD",
    "cryptoCurrency": "usdt_tron",
    "amount": 20,
    "payoutId": "po_xyz789",
    "payoutAmount": 19.50,
    "payoutCurrency": "USD",
    "completedAt": "2026-03-16T14:30:00.000Z"
  }
}
```

`ramp.failed` — payout submission failed:

```
{
  "event_type": "ramp.failed",
  "payment_data": {
    "merchant_id": "m_abc123",
    "intentId": "ri_1710345678000_a1b2c3d4e5f6g7h8",
    "type": "sell",
    "status": "failed",
    "fiatCurrency": "USD",
    "cryptoCurrency": "usdt_tron",
    "amount": 20,
    "reason": "Payout submission failed",
    "failedAt": "2026-03-16T14:30:00.000Z"
  }
}
```

Webhook vs Polling

We recommend using webhooks as the primary notification mechanism. Webhooks are delivered as soon as the ramp completes or fails — no delay. See the Webhooks guide for setup, signature verification, and retry behavior.

Complete Example

JAVASCRIPT

```javascript
// Full off-ramp flow
const runSellFlow = async () => {
  // Step 1: Create intent (server-side)
  const intent = await createSellIntent('customer@example.com', 'TLfMk...');
  console.log(`Intent: ${intent.intentId}`);

  // Step 2: Get quotes
  const quotes = await getSellQuotes(20);
  const selectedQuote = quotes[0];

  // Step 3: Phase 1 — create deposit
  const deposit = await createDeposit(intent.intentId, selectedQuote);
  console.log(`Send USDT to: ${deposit.depositAddress}`);

  // Step 4: Wait for deposit confirmation
  await waitForDeposit(intent.intentId);

  // Step 5: Phase 2a — get payout preview
  const preview = await getPayoutPreview(intent.intentId, 'USD', 'US');

  // Step 6: Phase 2b — submit payout (with customer's details)
  const payout = await submitPayout(intent.intentId, deposit.cryptoDepositId, {
    receiverFirstName: 'John',
    receiverLastName: 'Doe',
    receiverCountry: 'US',
    receiverAccountNumber: '123456789',
    receiverBankName: 'Bank of America',
    // ... other fields from preview.requiredFields
  });

  console.log(`Done! Payout ${payout.payoutId} is ${payout.status}`);
};
```

PYTHON

```python
def run_sell_flow():
    # Step 1: Create intent
    intent = create_sell_intent("customer@example.com", "TLfMk...")
    print(f"Intent: {intent['intentId']}")

    # Step 2: Get quotes
    quotes = get_sell_quotes(20)
    selected_quote = quotes[0]

    # Step 3: Phase 1 — create deposit
    deposit = create_deposit(intent["intentId"], selected_quote)

    # Step 4: Wait for deposit confirmation
    wait_for_deposit(intent["intentId"])

    # Step 5: Phase 2a — get payout preview
    preview = get_payout_preview(intent["intentId"], "USD", "US")

    # Step 6: Phase 2b — submit payout
    payout = submit_payout(intent["intentId"], deposit["cryptoDepositId"], {
        "receiverFirstName": "John",
        "receiverLastName": "Doe",
        "receiverCountry": "US",
        "receiverAccountNumber": "123456789",
        "receiverBankName": "Bank of America",
    })

    print(f"Done! Payout {payout['payoutId']} is {payout['status']}")

run_sell_flow()
```

Error Handling

Phase	Error	Recovery
Intent creation	Validation error	Check required fields and retry
Deposit (Phase 1)	Intent expired	Create a new intent
Deposit (Phase 1)	Deposit address generation failed	Retry — endpoint is idempotent
Waiting	Deposit not detected	Ensure correct chain and address; check with block explorer
Payout preview	Deposit not confirmed	Wait longer for blockchain confirmation
Payout (Phase 2)	FX quotation failed	Retry — the system will obtain a new quote
Payout (Phase 2)	Insufficient amount (fees > value)	Customer sent too little crypto

Next Steps

- Create Ramp Intent (https://docs.zenpays.com/docs/rest-api/endpoints/ramp/create-ramp-intent) — API reference
- Sell Quotes (https://docs.zenpays.com/docs/rest-api/endpoints/ramp/sell-quotes) — Quote endpoint details
- Create Sell Deposit (https://docs.zenpays.com/docs/rest-api/endpoints/ramp/sell-deposit) — Phase 1 reference
- Get Payout Preview (https://docs.zenpays.com/docs/rest-api/endpoints/ramp/payout-preview) — Phase 2a reference

- Submit Sell Payout (https://docs.zenpays.com/docs/rest-api/endpoints/ramp/sell-payout) — Phase 2b reference
- On-Ramp Flow — Buy crypto integration guide

EXAMPLES

Payout Flow

Send money to customers and vendors using the two-step intent flow.

Payout Intent Flow (Recommended)

JAVASCRIPT

```javascript
import { ZenPays } from '@zenxdigitalholdings/zenpays'

async function payoutWithIntent() {
  const zenpays = new ZenPays({
    apiKey: process.env.ZENPAYS_API_KEY!,
  })

  // Step 1: Create payout intent with basic details
  const intent = await zenpays.payouts.createPayoutIntent({
    amount: 10000,
    currency: 'INR',
    country: 'IN',
    beneficiaryName: 'John Doe',
    beneficiaryEmail: 'john@example.com',
    purpose: 'SALARY',
  })

  console.log('Intent created:', intent.intentId)
  console.log('Status:', intent.status) // 'requires_confirmation'
  console.log('Expires at:', intent.expiresAt)

  // The response tells you exactly what fields to collect
  console.log('Required fields:')
  intent.requiredFields?.forEach(field => {
    console.log(`  - ${field.label} (${field.fieldName}) [${field.type}]`)
  })

  // Step 2: Confirm with the required fields
  const result = await zenpays.payouts.confirmPayoutIntent(intent.intentId, {
    beneficiaryAccount: '1234567890',
    beneficiaryIfsc: 'HDFC0001234',
  })

  console.log('Payout confirmed:', result.payoutId)
  console.log('Status:', result.status) // 'processing'

  // Step 3: Poll for completion
  const poll = async (id: string): Promise<void> => {
    const current = await zenpays.payouts.getPayoutIntent(id)

    if (current.status === 'succeeded') {
      console.log('Payout completed!')
      return
    }
    if (current.status === 'failed') {
      console.log('Payout failed:', current.failureReason)
      return
    }

    console.log('Status:', current.status)
    await new Promise(r => setTimeout(r, 5000))
    return poll(id)
  }

  await poll(intent.intentId)
}

// Crypto payout example
async function cryptoPayout() {
  const zenpays = new ZenPays({
```

```
    apiKey: process.env.ZENPAYS_API_KEY!,
  })

  // Step 1: Create intent for crypto (USDT)
  const intent = await zenpays.payouts.createPayoutIntent({
    amount: 500,
    currency: 'USDT',
    beneficiaryName: 'Crypto Wallet',
  })

  // For crypto, requiredFields will include "chain" and "walletAddress"
  console.log(intent.requiredFields)
  // [{ fieldName: "chain", type: "select", options: ["ethereum", "tron", ...] },
  //  { fieldName: "walletAddress", type: "text" }]

  // Step 2: Confirm with chain and address
  const result = await zenpays.payouts.confirmPayoutIntent(intent.intentId, {
    chain: 'tron',
    walletAddress: 'TXyz123abc...',
  })

  console.log('Crypto payout processing:', result.payoutId)
}

payoutWithIntent()
```

PYTHON

```python
import os
import time
from zenpays import ZenPays


def payout_with_intent():
    zenpays = ZenPays(api_key=os.environ["ZENPAYS_API_KEY"])

    # Step 1: Create payout intent
    intent = zenpays.payouts.create_payout_intent({
        "amount": 10000,
        "currency": "INR",
        "country": "IN",
        "beneficiary_name": "John Doe",
        "beneficiary_email": "john@example.com",
        "purpose": "SALARY",
    })

    print(f"Intent created: {intent['intent_id']}")
    print(f"Status: {intent['status']}")  # "requires_confirmation"

    # The response tells you what fields to collect
    for field in intent["required_fields"]:
        print(f"  - {field['label']} ({field['field_name']})")

    # Step 2: Confirm with the required fields
    result = zenpays.payouts.confirm_payout_intent(intent["intent_id"], {
        "beneficiary_account": "1234567890",
        "beneficiary_ifsc": "HDFC0001234",
    })

    print(f"Payout confirmed: {result['payout_id']}")
    print(f"Status: {result['status']}")  # "processing"

    # Step 3: Poll for completion
    def poll(intent_id):
        current = zenpays.payouts.get_payout_intent(intent_id)
        if current["status"] == "succeeded":
            print("Payout completed!")
            return
        if current["status"] == "failed":
            print(f"Payout failed: {current.get('failure_reason')}")
            return
        print(f"Status: {current['status']}")
        time.sleep(5)
        poll(intent_id)

    poll(intent["intent_id"])


def crypto_payout():
    zenpays = ZenPays(api_key=os.environ["ZENPAYS_API_KEY"])

    # Step 1: Crypto intent
    intent = zenpays.payouts.create_payout_intent({
        "amount": 500,
        "currency": "USDT",
        "beneficiary_name": "Crypto Wallet",
    })

    # Step 2: Confirm with chain + address
    result = zenpays.payouts.confirm_payout_intent(intent["intent_id"], {
```

```
        "chain": "tron",
        "wallet_address": "TXyz123abc...",
    })

    print(f"Crypto payout processing: {result['payout_id']}")


if __name__ == "__main__":
    payout_with_intent()
```

Legacy Direct Payout

> **Deprecated**
> The direct payout API is deprecated. Use the Payout Intent flow above instead.

JAVASCRIPT

```javascript
import { ValidationError, ZenPays } from '@zenxdigitalholdings/zenpays'

async function processPayout() {
  const zenpays = new ZenPays({
    apiKey: process.env.ZENPAYS_API_KEY!,
  })

  // 1. Check available payout methods
  const methods = await zenpays.payouts.getMethods()
  console.log('Available payout methods:', methods.map(m => m.type).join(', '))

  // 2. Check wallet balance
  const balance = await zenpays.wallet.getBalance('USD')
  console.log('Available balance:', balance)

  // 3. Create payout
  const payout = await zenpays.payouts.create({
    customerId: 'cust_xxx',
    amount: 10000, // $100.00
    currency: 'USD',
    payoutMethod: 'bank_transfer',
    beneficiaryDetails: {
      name: 'John Doe',
      email: 'john@example.com',
      bankName: 'Chase Bank',
      accountNumber: '123456789',
      accountType: 'checking',
      routingNumber: '021000021',
    },
    description: 'Withdrawal request',
    idempotencyKey: 'payout_unique_123', // Prevent duplicates
  })

  console.log('Payout created:', payout.id)
  console.log('Status:', payout.status)

  // 4. Monitor payout status
  const waitForPayout = async (payoutId: string): Promise<boolean> => {
    const p = await zenpays.payouts.get(payoutId)

    switch (p.status) {
      case 'completed':
        console.log('Payout completed!')
        console.log('Processed at:', p.processedAt)
        return true

      case 'failed':
        console.log('Payout failed:', p.failureReason)
        return false

      case 'processing':
        console.log('Payout processing...')
        break

      default:
        console.log('Payout status:', p.status)
    }

    await new Promise(resolve => setTimeout(resolve, 5000))
    return waitForPayout(payoutId)
  }
```

```
  return waitForPayout(payout.id)
}

// Retry a failed payout
async function retryFailedPayout(payoutId: string) {
  const zenpays = new ZenPays({
    apiKey: process.env.ZENPAYS_API_KEY!,
  })

  const payout = await zenpays.payouts.get(payoutId)

  if (payout.status !== 'failed') {
    console.log('Payout is not failed, cannot retry')
    return
  }

  console.log('Retrying payout:', payoutId)
  const retried = await zenpays.payouts.retry(payoutId)
  console.log('Retry status:', retried.status)
}

// Get customer payouts
async function getCustomerPayouts(customerId: string) {
  const zenpays = new ZenPays({
    apiKey: process.env.ZENPAYS_API_KEY!,
  })

  const { data, total } = await zenpays.payouts.listByCustomer(customerId, {
    limit: 10,
  })

  console.log(`Found ${total} payouts for customer ${customerId}:`)
  data.forEach((p) => {
    console.log(`  ${p.id}: ${p.amount} ${p.currency} - ${p.status}`)
  })
}

processPayout()
```

PYTHON

```python
import os
import time
from zenpays import ZenPays
from zenpays.errors import ValidationError


def process_payout():
    zenpays = ZenPays(api_key=os.environ["ZENPAYS_API_KEY"])

    # 1. Check available payout methods
    methods = zenpays.payouts.get_methods()
    print(f"Available payout methods: {', '.join(m['type'] for m in methods)}")

    # 2. Check wallet balance
    balance = zenpays.wallet.get_balance("USD")
    print(f"Available balance: {balance}")

    # 3. Create payout
    payout = zenpays.payouts.create({
        "customer_id": "cust_xxx",
        "amount": 10000,  # $100.00
        "currency": "USD",
        "payout_method": "bank_transfer",
        "beneficiary_details": {
            "name": "John Doe",
            "email": "john@example.com",
            "bank_name": "Chase Bank",
            "account_number": "123456789",
            "account_type": "checking",
            "routing_number": "021000021",
        },
        "description": "Withdrawal request",
        "idempotency_key": "payout_unique_123",  # Prevent duplicates
    })

    print(f"Payout created: {payout['id']}")
    print(f"Status: {payout['status']}")

    # 4. Monitor payout status
    def wait_for_payout(payout_id: str) -> bool:
        p = zenpays.payouts.get(payout_id)

        status = p["status"]

        if status == "completed":
            print("Payout completed!")
            print(f"Processed at: {p.get('processed_at')}")
            return True

        elif status == "failed":
            print(f"Payout failed: {p.get('failure_reason')}")
            return False

        elif status == "processing":
            print("Payout processing...")

        else:
            print(f"Payout status: {status}")

        time.sleep(5)
        return wait_for_payout(payout_id)
```

```
    return wait_for_payout(payout["id"])


# Retry a failed payout
def retry_failed_payout(payout_id: str):
    zenpays = ZenPays(api_key=os.environ["ZENPAYS_API_KEY"])

    payout = zenpays.payouts.get(payout_id)

    if payout["status"] != "failed":
        print("Payout is not failed, cannot retry")
        return

    print(f"Retrying payout: {payout_id}")
    retried = zenpays.payouts.retry(payout_id)
    print(f"Retry status: {retried['status']}")


# Get customer payouts
def get_customer_payouts(customer_id: str):
    zenpays = ZenPays(api_key=os.environ["ZENPAYS_API_KEY"])

    result = zenpays.payouts.list_by_customer(customer_id, {"limit": 10})
    data = result["data"]
    total = result["total"]

    print(f"Found {total} payouts for customer {customer_id}:")
    for p in data:
        print(f"  {p['id']}: {p['amount']} {p['currency']} - {p['status']}")


if __name__ == "__main__":
    process_payout()
```

Cancel a Payout

Cancel a payout that hasn't started processing yet.

JAVASCRIPT

```
import { ZenPays } from '@zenxdigitalholdings/zenpays'

async function cancelPayout(payoutId: string) {
  const zenpays = new ZenPays({
    apiKey: process.env.ZENPAYS_API_KEY!,
  })

  // Check current status
  const payout = await zenpays.payouts.get(payoutId)

  if (payout.status !== 'pending') {
    console.log(`Cannot cancel — payout is already ${payout.status}`)
    return
  }

  // Cancel the payout
  const cancelled = await zenpays.payouts.cancel(payoutId, 'Customer requested cancellation')
  console.log('Cancelled:', cancelled.status) // 'cancelled'
}

cancelPayout('pay_xxx')
```

PYTHON

```python
import os
from zenpays import ZenPays


def cancel_payout(payout_id: str):
    zenpays = ZenPays(api_key=os.environ["ZENPAYS_API_KEY"])

    # Check current status
    payout = zenpays.payouts.get(payout_id)

    if payout["status"] != "pending":
        print(f"Cannot cancel — payout is already {payout['status']}")
        return

    # Cancel the payout
    cancelled = zenpays.payouts.cancel(payout_id, "Customer requested cancellation")
    print(f"Cancelled: {cancelled['status']}")  # "cancelled"


if __name__ == "__main__":
    cancel_payout("pay_xxx")
```

Batch Payout Flow

Send payouts to multiple beneficiaries in a single batch.

JAVASCRIPT

```javascript
import { ZenPays } from '@zenxdigitalholdings/zenpays'

async function processBatchPayout() {
  const zenpays = new ZenPays({
    apiKey: process.env.ZENPAYS_API_KEY!,
  })

  // 1. Create batch payout
  const batch = await zenpays.payouts.createBatch({
    items: [
      {
        beneficiaryName: 'Alice Smith',
        beneficiaryAccount: '1234567890',
        beneficiaryIfsc: 'HDFC0001234',
        amount: 5000,
        currency: 'INR',
        payoutType: 'imps',
        purpose: 'SALARY',
      },
      {
        beneficiaryName: 'Bob Johnson',
        beneficiaryVpa: 'bob@upi',
        amount: 3000,
        currency: 'INR',
        payoutType: 'upi',
        purpose: 'COMMISSION',
      },
      {
        beneficiaryName: 'Carol Williams',
        beneficiaryAccount: '9876543210',
        beneficiaryIfsc: 'ICIC0004567',
        amount: 7000,
        currency: 'INR',
        payoutType: 'neft',
        purpose: 'PAYOUT',
      },
    ],
    currency: 'INR',
    webhookUrl: 'https://example.com/webhooks/batch',
  })

  console.log('Batch created:', batch.batchId)
  console.log('Total amount:', batch.totalAmount, batch.currency)
  console.log('Estimated fees:', batch.estimatedFees)
  console.log('Status:', batch.status) // 'validated'

  // 2. Confirm the batch to start processing
  const confirmed = await zenpays.payouts.confirmBatch(batch.batchId)
  console.log('Processing started:', confirmed.status) // 'processing'

  // 3. Poll for completion
  const waitForBatch = async (batchId: string): Promise<void> => {
    const status = await zenpays.payouts.getBatch(batchId)
    console.log(`Progress: ${status.progressPercentage}% (${status.successfulPayouts} succeeded,
${status.failedPayouts} failed)`)

    if (status.status === 'completed' || status.status === 'partially_completed') {
      console.log('Batch finished!')

      // 4. Check individual results
      const { data: payouts } = await zenpays.payouts.getBatchPayouts(batchId)
      payouts.forEach(p => {
```

```
        console.log(`  ${p.id}: ${p.amount} ${p.currency} → ${p.status}`)
      })

      // 5. Check for failures
      const { data: failed } = await zenpays.payouts.getBatchPayouts(batchId, { status: 'failed'
})
      if (failed.length > 0) {
        console.log(`${failed.length} payouts failed:`)
        failed.forEach(p => console.log(`  ${p.id}: ${p.failureReason}`))
      }
      return
    }

    await new Promise(resolve => setTimeout(resolve, 5000))
    return waitForBatch(batchId)
  }

  await waitForBatch(batch.batchId)
}

processBatchPayout()
```

PYTHON

```python
import os
import time
from zenpays import ZenPays


def process_batch_payout():
    zenpays = ZenPays(api_key=os.environ["ZENPAYS_API_KEY"])

    # 1. Create batch payout
    batch = zenpays.payouts.create_batch({
        "items": [
            {
                "beneficiary_name": "Alice Smith",
                "beneficiary_account": "1234567890",
                "beneficiary_ifsc": "HDFC0001234",
                "amount": 5000,
                "currency": "INR",
                "payout_type": "imps",
                "purpose": "SALARY",
            },
            {
                "beneficiary_name": "Bob Johnson",
                "beneficiary_vpa": "bob@upi",
                "amount": 3000,
                "currency": "INR",
                "payout_type": "upi",
                "purpose": "COMMISSION",
            },
            {
                "beneficiary_name": "Carol Williams",
                "beneficiary_account": "9876543210",
                "beneficiary_ifsc": "ICIC0004567",
                "amount": 7000,
                "currency": "INR",
                "payout_type": "neft",
                "purpose": "PAYOUT",
            },
        ],
        "currency": "INR",
        "webhook_url": "https://example.com/webhooks/batch",
    })

    print(f"Batch created: {batch['batch_id']}")
    print(f"Total amount: {batch['total_amount']} {batch['currency']}")
    print(f"Status: {batch['status']}")  # "validated"

    # 2. Confirm the batch to start processing
    confirmed = zenpays.payouts.confirm_batch(batch["batch_id"])
    print(f"Processing started: {confirmed['status']}")  # "processing"

    # 3. Poll for completion
    def wait_for_batch(batch_id: str):
        status = zenpays.payouts.get_batch(batch_id)
        print(f"Progress: {status['progress_percentage']}% "
              f"({status['successful_payouts']} succeeded, {status['failed_payouts']} failed)")

        if status["status"] in ("completed", "partially_completed"):
            print("Batch finished!")

            # 4. Check individual results
            result = zenpays.payouts.get_batch_payouts(batch_id)
            for p in result["data"]:
```

```
                print(f"  {p['id']}: {p['amount']} {p['currency']} -> {p['status']}")

            # 5. Check for failures
            failed = zenpays.payouts.get_batch_payouts(batch_id, {"status": "failed"})
            if failed["data"]:
                print(f"{len(failed['data'])} payouts failed:")
                for p in failed["data"]:
                    print(f"  {p['id']}: {p.get('failure_reason')}")
            return

        time.sleep(5)
        wait_for_batch(batch_id)

    wait_for_batch(batch["batch_id"])


if __name__ == "__main__":
    process_batch_payout()
```

EXAMPLES

Refund Flow

Process INR refunds -- full refunds, partial refunds, and batch operations. Refunds are paid out to the customer's bank account via IMPS.

JAVASCRIPT

```javascript
import { ZenPays } from '@zenxdigitalholdings/zenpays'

const zenpays = new ZenPays({ apiKey: process.env.ZENPAYS_API_KEY! })

// ---------------------------------------------------------------------------
// 1. Full refund
// ---------------------------------------------------------------------------
async function fullRefund(transactionId: string) {
  const refund = await zenpays.refunds.create({
    transactionId,
    reason: 'Customer requested full refund',
    beneficiaryEmail: 'priya@example.com',
    beneficiaryAccount: '50100012345678',
    beneficiaryIfsc: 'HDFC0001234',
  })

  console.log('Refund created:', refund.refundId)
  console.log('Amount:', refund.amount, refund.currency)
  console.log('Status:', refund.status) // pending_approval
}

// ---------------------------------------------------------------------------
// 2. Partial refund
// ---------------------------------------------------------------------------
async function partialRefund(transactionId: string) {
  const refund = await zenpays.refunds.create({
    transactionId,
    amount: 500, // Refund 500 INR out of the total
    reason: 'Partial item return',
    beneficiaryEmail: 'priya@example.com',
    beneficiaryAccount: '50100012345678',
    beneficiaryIfsc: 'HDFC0001234',
  })

  console.log('Partial refund:', refund.refundId)
  console.log('Refund amount:', refund.amount) // 500
  console.log('Original amount:', refund.originalAmount) // e.g. 2000
  console.log('Type:', refund.refundType) // partial
}

// ---------------------------------------------------------------------------
// 3. Refund with beneficiary name
// ---------------------------------------------------------------------------
async function refundViaBankAccount(transactionId: string) {
  const refund = await zenpays.refunds.create({
    transactionId,
    amount: 750,
    reason: 'Order cancelled',
    beneficiaryEmail: 'rahul@example.com',
    beneficiaryAccount: '50100012345678',
    beneficiaryIfsc: 'HDFC0001234',
    beneficiaryName: 'Rahul Sharma',
  })

  console.log('Bank refund:', refund.refundId)
  console.log('Status:', refund.status)
}

// ---------------------------------------------------------------------------
// 4. Multiple partial refunds on the same transaction
// ---------------------------------------------------------------------------
async function progressivePartialRefunds(transactionId: string) {
```

```
  // Original transaction: 1000 INR

  // First partial refund: 300 INR
  const refund1 = await zenpays.refunds.create({
    transactionId,
    amount: 300,
    reason: 'Damaged item',
    beneficiaryEmail: 'customer@example.com',
    beneficiaryAccount: '91020034567890',
    beneficiaryIfsc: 'SBIN0001234',
  })
  console.log(`Refund 1: ${refund1.refundId}, amount: 300`)
  // Transaction: refundedAmount = 300, status = "partial_refund"

  // Second partial refund: 200 INR
  const refund2 = await zenpays.refunds.create({
    transactionId,
    amount: 200,
    reason: 'Shipping overcharge',
    beneficiaryEmail: 'customer@example.com',
    beneficiaryAccount: '91020034567890',
    beneficiaryIfsc: 'SBIN0001234',
  })
  console.log(`Refund 2: ${refund2.refundId}, amount: 200`)
  // Transaction: refundedAmount = 500, status = "partial_refund"

  // Final refund: remaining 500 INR
  const refund3 = await zenpays.refunds.create({
    transactionId,
    amount: 500,
    reason: 'Remaining balance',
    beneficiaryEmail: 'customer@example.com',
    beneficiaryAccount: '91020034567890',
    beneficiaryIfsc: 'SBIN0001234',
  })
  console.log(`Refund 3: ${refund3.refundId}, amount: 500`)
  // Transaction: refundedAmount = 1000, status = "refunded"

  // This would fail -- already fully refunded:
  // await zenpays.refunds.create({ transactionId, amount: 1, ... })
  // Error: "Transaction has already been fully refunded"
}

// ---------------------------------------------------------------------------
// 5. Batch refund
// ---------------------------------------------------------------------------
async function batchRefund() {
  const result = await zenpays.refunds.bulkUpload({
    refunds: [
      {
        transactionId: 'txn_aaa',
        reason: 'Customer request',
        beneficiaryEmail: 'john@example.com',
        beneficiaryAccount: '30200045678901',
        beneficiaryIfsc: 'ICIC0001234',
      },
      {
        transactionId: 'txn_bbb',
        amount: 500,
        reason: 'Partial refund',
        beneficiaryEmail: 'jane@example.com',
        beneficiaryAccount: '1234567890',
        beneficiaryIfsc: 'HDFC0001234',
      },
    ],
```

```
  })

  console.log('Total:', result.total)
  console.log('Succeeded:', result.succeeded)
  console.log('Failed:', result.failed)

  result.results.forEach((r) => {
    if (r.status === 'success') {
      console.log(`  ${r.transactionId}: Refund ${r.refundId}`)
    } else {
      console.log(`  ${r.transactionId}: Failed - ${r.error}`)
    }
  })
}

// ---------------------------------------------------------------------------
// 6. Poll for refund status
// ---------------------------------------------------------------------------
async function waitForRefund(refundId: string): Promise<string> {
  const refund = await zenpays.refunds.get(refundId)

  switch (refund.status) {
    case 'completed':
      console.log('Refund completed successfully')
      return 'completed'

    case 'failed':
      console.log('Refund failed:', refund.failureReason)
      return 'failed'

    case 'rejected':
      console.log('Refund rejected')
      return 'rejected'

    case 'cancelled':
      console.log('Refund cancelled')
      return 'cancelled'

    default:
      // pending_approval, approved, processing -- keep polling
      console.log('Refund status:', refund.status)
      await new Promise(resolve => setTimeout(resolve, 5000))
      return waitForRefund(refundId)
  }
}

// ---------------------------------------------------------------------------
// 7. Get refund statistics
// ---------------------------------------------------------------------------
async function getRefundStats() {
  const stats = await zenpays.refunds.getStats('2026-01-01', '2026-12-31', 'INR')

  console.log('Refund Statistics:')
  console.log('  Total refunds:', stats.totalCount)
  console.log('  Total amount:', stats.totalAmount, 'INR')
  console.log('  Pending:', stats.pendingCount)
  console.log('  Completed:', stats.completedCount)
  console.log('  Failed:', stats.failedCount)
}
```

PYTHON

```python
import os
import time
from zenpays import ZenPays

zenpays = ZenPays(api_key=os.environ["ZENPAYS_API_KEY"])


# ---------------------------------------------------------------------------
# 1. Full refund
# ---------------------------------------------------------------------------
def full_refund(transaction_id: str):
    refund = zenpays.refunds.create({
        "transactionId": transaction_id,
        "reason": "Customer requested full refund",
        "beneficiaryEmail": "priya@example.com",
        "beneficiaryAccount": "50100012345678",
        "beneficiaryIfsc": "HDFC0001234",
    })

    print(f"Refund created: {refund['refundId']}")
    print(f"Amount: {refund['amount']} {refund['currency']}")
    print(f"Status: {refund['status']}")  # pending_approval


# ---------------------------------------------------------------------------
# 2. Partial refund
# ---------------------------------------------------------------------------
def partial_refund(transaction_id: str):
    refund = zenpays.refunds.create({
        "transactionId": transaction_id,
        "amount": 500,
        "reason": "Partial item return",
        "beneficiaryEmail": "priya@example.com",
        "beneficiaryAccount": "50100012345678",
        "beneficiaryIfsc": "HDFC0001234",
    })

    print(f"Partial refund: {refund['refundId']}")
    print(f"Refund amount: {refund['amount']}")  # 500
    print(f"Original amount: {refund['originalAmount']}")  # e.g. 2000
    print(f"Type: {refund['refundType']}")  # partial


# ---------------------------------------------------------------------------
# 3. Refund with beneficiary name
# ---------------------------------------------------------------------------
def refund_via_bank_account(transaction_id: str):
    refund = zenpays.refunds.create({
        "transactionId": transaction_id,
        "amount": 750,
        "reason": "Order cancelled",
        "beneficiaryEmail": "rahul@example.com",
        "beneficiaryAccount": "50100012345678",
        "beneficiaryIfsc": "HDFC0001234",
        "beneficiaryName": "Rahul Sharma",
    })

    print(f"Bank refund: {refund['refundId']}")
    print(f"Status: {refund['status']}")


# ---------------------------------------------------------------------------
```

```python
# 4. Multiple partial refunds on the same transaction
# ---------------------------------------------------------------------------
def progressive_partial_refunds(transaction_id: str):
    # Original transaction: 1000 INR

    refund1 = zenpays.refunds.create({
        "transactionId": transaction_id,
        "amount": 300,
        "reason": "Damaged item",
        "beneficiaryEmail": "customer@example.com",
        "beneficiaryAccount": "91020034567890",
        "beneficiaryIfsc": "SBIN0001234",
    })
    print(f"Refund 1: {refund1['refundId']}, amount: 300")
    # Transaction: refundedAmount = 300, status = "partial_refund"

    refund2 = zenpays.refunds.create({
        "transactionId": transaction_id,
        "amount": 200,
        "reason": "Shipping overcharge",
        "beneficiaryEmail": "customer@example.com",
        "beneficiaryAccount": "91020034567890",
        "beneficiaryIfsc": "SBIN0001234",
    })
    print(f"Refund 2: {refund2['refundId']}, amount: 200")
    # Transaction: refundedAmount = 500, status = "partial_refund"

    refund3 = zenpays.refunds.create({
        "transactionId": transaction_id,
        "amount": 500,
        "reason": "Remaining balance",
        "beneficiaryEmail": "customer@example.com",
        "beneficiaryAccount": "91020034567890",
        "beneficiaryIfsc": "SBIN0001234",
    })
    print(f"Refund 3: {refund3['refundId']}, amount: 500")
    # Transaction: refundedAmount = 1000, status = "refunded"


# ---------------------------------------------------------------------------
# 5. Batch refund
# ---------------------------------------------------------------------------
def batch_refund():
    result = zenpays.refunds.bulk_upload({
        "refunds": [
            {
                "transactionId": "txn_aaa",
                "reason": "Customer request",
                "beneficiaryEmail": "john@example.com",
                "beneficiaryAccount": "30200045678901",
                "beneficiaryIfsc": "ICIC0001234",
            },
            {
                "transactionId": "txn_bbb",
                "amount": 500,
                "reason": "Partial refund",
                "beneficiaryEmail": "jane@example.com",
                "beneficiaryAccount": "1234567890",
                "beneficiaryIfsc": "HDFC0001234",
            },
        ],
    })

    print(f"Total: {result['total']}")
    print(f"Succeeded: {result['succeeded']}")
```

```
    print(f"Failed: {result['failed']}")

    for r in result["results"]:
        if r["status"] == "success":
            print(f"  {r['transactionId']}: Refund {r['refundId']}")
        else:
            print(f"  {r['transactionId']}: Failed - {r['error']}")


# ---------------------------------------------------------------------------
# 6. Poll for refund status
# ---------------------------------------------------------------------------
def wait_for_refund(refund_id: str) -> str:
    refund = zenpays.refunds.get(refund_id)

    if refund["status"] == "completed":
        print("Refund completed successfully")
        return "completed"

    if refund["status"] == "failed":
        print(f"Refund failed: {refund.get('failureReason')}")
        return "failed"

    if refund["status"] in ("rejected", "cancelled"):
        print(f"Refund {refund['status']}")
        return refund["status"]

    # pending_approval, approved, processing -- keep polling
    print(f"Refund status: {refund['status']}")
    time.sleep(5)
    return wait_for_refund(refund_id)


# ---------------------------------------------------------------------------
# 7. Get refund statistics
# ---------------------------------------------------------------------------
def get_refund_stats():
    stats = zenpays.refunds.get_stats("2026-01-01", "2026-12-31", "INR")

    print("Refund Statistics:")
    print(f"  Total refunds: {stats['totalCount']}")
    print(f"  Total amount: {stats['totalAmount']} INR")
    print(f"  Pending: {stats['pendingCount']}")
    print(f"  Completed: {stats['completedCount']}")
    print(f"  Failed: {stats['failedCount']}")
```

PART

Resources

2 documents

RESOURCES

Postman Collection

Postman Collection

Test and explore the ZenPays API using our pre-configured Postman collection with 60+ API endpoints across 10 categories.

Download

Download the Postman collection (JSON)

Features

- **60+ Pre-configured Endpoints** - All API endpoints ready to use
- **Auto HMAC Signing** - Pre-request script auto-generates `X-Signature` and `X-Timestamp` headers
- **Example Request Bodies** - Realistic sample data for every endpoint
- **Environment Variables** - Easy switching between sandbox and production
- **Optional Query Params** - Disabled filters you can toggle on as needed

Included Endpoints

Category	Endpoints	Description
Payment Intents	3	Create, get, and confirm payments
Transactions	4	List, search, and export
Customers	7	Create, update, risk, top customers
Payouts	8	Create, preview, retry, stats, export
Refunds	5	Create, list, cancel, stats
Settlements	9	Create, cancel, bank accounts management
Wallet	4	Balances, transactions, summary
Ledger	8	Entries, balances, reserves, reconciliation
Reports	5	Generate, list, download, delete
Health	1	Health check

Quick Start

1. **Download** the collection using the button above
2. **Open Postman** and click "Import"
3. **Select** the downloaded JSON file
4. **Set variables**: `apiKey` and `secretSalt` from your ZenPays Dashboard
5. **Start testing** - signatures are generated automatically!

Collection Variables

Variable	Description
`baseUrl`	API base URL (default: `https://api.zenpays.com`)
`apiKey`	Your API key (e.g., `zp_test_xxxxx` or `zp_live_xxxxx`)
`secretSalt`	Your secret salt for HMAC signature generation

Authentication

All requests use **API Key + HMAC Signature** authentication. The collection includes a **pre-request script** that automatically generates all required headers before every request — you just need to set your credentials once.

Required Headers (auto-generated)

Every request sends these 5 headers, all handled automatically by the pre-request script:

Header	Value	Description
`Authorization`	`Bearer {apiKey}`	Your API key from the collection variables
`X-Timestamp`	ISO 8601 UTC timestamp	Generated at request time (e.g., `2024-01-15T10:30:00.000Z`)
`X-Signature`	HMAC-SHA256 hex string	Computed from timestamp + body using your secret salt
`X-Secret-Salt`	Your secret salt	Passed for server-side HMAC validation
`Content-Type`	`application/json`	Set on all requests

How the Pre-Request Script Works

The collection's pre-request script runs before every request and does the following:

```
// 1. Reads your credentials from collection variables
const apiKey = pm.collectionVariables.get('apiKey');
const secretSalt = pm.collectionVariables.get('secretSalt');

// 2. Generates a fresh timestamp
const timestamp = new Date().toISOString();

// 3. Compacts the request body (strips whitespace/newlines)
//    For GET requests or empty bodies, uses '{}'
const body = pm.request.body?.raw
  ? JSON.stringify(JSON.parse(pm.request.body.raw))
  : '{}';

// 4. Computes HMAC-SHA256 signature
const signature = CryptoJS.HmacSHA256(timestamp + body, secretSalt).toString();

// 5. Sets all required headers automatically
pm.request.headers.upsert({ key: 'Authorization', value: 'Bearer ' + apiKey });
pm.request.headers.upsert({ key: 'X-Timestamp', value: timestamp });
pm.request.headers.upsert({ key: 'X-Signature', value: signature });
pm.request.headers.upsert({ key: 'X-Secret-Salt', value: secretSalt });
pm.request.headers.upsert({ key: 'Content-Type', value: 'application/json' });
```

Important

The `apiKey` collection variable should contain **only** the key itself (e.g., `zp_test_abc123`), **not** the `Bearer` prefix. The script adds `Bearer` automatically.

Environment Setup

Option 1: Collection Variables (Recommended)

After importing, set the variables directly in the collection:

1. Click the collection name **"ZenPays API"** in the sidebar
2. Go to the **Variables** tab
3. Set these values in the **Current Value** column:

Variable	Current Value
`baseUrl`	`https://api.zenpays.com`
`apiKey`	`zp_test_your_key_here`
`secretSalt`	`your_secret_salt_here`

4. Click **Save**

Option 2: Postman Environments

Create a Postman Environment for each stage:

1. Click the **Environments** tab (gear icon) in the sidebar
2. Click **+** to create a new environment
3. Name it (e.g., "ZenPays Sandbox")
4. Add the same 3 variables: `baseUrl`, `apiKey`, `secretSalt`

5. Select the environment from the dropdown in the top-right corner

> **Tip**
> If you use **both** environment variables and collection variables, environment variables take priority. Make sure to clear any environment overrides if you want the collection variables to be used.

Environments

Environment	Base URL
Sandbox	`https://api.zenpays.com`

Troubleshooting

Issue	Cause	Fix
`401 Missing Authorization header`	Pre-request script not running	Re-import the collection; ensure `apiKey` and `secretSalt` are set
`401 Invalid HMAC signature`	Body format mismatch or wrong secret salt	Verify `secretSalt` is correct; check Postman Console for sent headers
`Bearer Bearer zp_test_...` in logs	`apiKey` variable includes `Bearer` prefix	Remove `Bearer` from the `apiKey` value — the script adds it
Headers not appearing on request	Environment variable overrides with empty values	Clear or delete environment variable overrides
`Request timestamp expired`	System clock drift or stale request	Ensure your system clock is accurate; the signature expires after 5 minutes

Tips

- **View Console** - Use Postman Console (**View → Show Postman Console**) to see the exact headers being sent and debug authentication issues
- **Toggle filters** - Many GET endpoints have optional query parameters (disabled by default) — enable the ones you need
- **Idempotency** - POST endpoints include `X-Idempotency-Key` headers to prevent duplicate requests
- **Use environments** - Create separate Postman environments for sandbox vs production to avoid accidental production calls

RESOURCES

Using ZenPays docs with AI tools

Every page on this site is published as plain Markdown as well as HTML, so a coding agent can read the real source instead of scraping a rendered page.

Copy one page

Every page has a **Copy page** button at the top right. It copies that page as Markdown, ready to paste into ChatGPT, Claude, Cursor or Copilot Chat.

View as Markdown opens the same content as a plain file, which is the URL to give a tool that fetches its own context.

Fetch a page directly

Append the page's path to `/md/` and add `.md`:

```
# The HTML page
https://docs.zenpays.com/docs/rest-api/endpoints/payments/create-payment-intent

# The same page as Markdown
curl https://docs.zenpays.com/md/rest-api/endpoints/payments/create-payment-intent.md
```

The Markdown is the page's own source with the site's plumbing removed. Code samples keep their language labels, so a model can tell the cURL example from the TypeScript one.

Point an agent at the whole site

`llms.txt` is an index of every page, grouped by section, linking to the Markdown rather than the HTML:

```
curl https://docs.zenpays.com/llms.txt
```

Give an agent that one URL and it can find and fetch any page it needs.

Cursor

Settings → **Features** → **Docs** → **Add new doc**, and enter:

```
https://docs.zenpays.com/llms.txt
```

Then reference it in chat with `@ZenPays`.

Claude Code

Point Claude at the index in a prompt, or add it to your project's `CLAUDE.md` so it is always available:

```
ZenPays API reference: https://docs.zenpays.com/llms.txt
Fetch any page as Markdown from https://docs.zenpays.com/md/<path>.md
```

Anything that fetches URLs

Give it `https://docs.zenpays.com/llms.txt` and it can navigate from there. No key, no account, no rate limit.

Offline copies

For a model with a large context window, or for a reader with no network, the PDF downloads contain the entire documentation set as four volumes. The complete edition is a single file covering every published page.

> **Tip**
> When you ask an AI to write ZenPays integration code, paste the specific endpoint page rather than describing the endpoint. Amounts in particular are easy to get wrong from memory — every `amount` is an integer in the currency's smallest unit, and the divisor is not always 100.