

DEVELOPER DOCUMENTATION

Integration guide

QR charges and payouts API for Paraguay

Contents

1	Introduction	3
1.1	Base URL	3
1.2	Keys	3
1.3	Viewing the secret again	3
2	Quickstart — first charge in 5 minutes	4
2.1	1. Install the SDK	4
2.2	2. A route that creates the charge with the amount from your database	4
2.3	3. Embed the checkout with bepago.js	5
2.4	4. Receive the webhook with the SDK handler	6
3	Official SDKs	8
3.1	Node.js (18+, no dependencies)	8
3.2	PHP (7.4+, no dependencies)	8
3.3	Other languages	9
3.4	Official example	9
4	Authentication	10
4.1	Required headers on every request	10
4.2	Canonical string	10
4.3	Implementation — Node	11
4.4	Implementation — PHP	11
4.5	Implementation — Python	11
4.6	Test vector	12
5	Charges	13
5.1	Create a charge	13
5.2	Retrieve a charge	13
5.3	List charges	13
5.4	Refund	14
5.5	Cancel	14
5.6	Collected balance	14
5.7	Charge statuses	15
6	Embedded checkout	16
6.1	Embed with bepago.js (recommended)	16
6.2	Allowed domains	16
6.3	Payment methods	16
7	Payouts	18

7.1	Create a payout	18
7.2	Bank account destination	18
7.3	Alias destination	18
7.4	Retrieve	18
7.5	List payouts	19
7.6	Cancel	19
7.7	Balance	19
7.8	Statuses	19
7.9	Batches (bulk payouts)	20
8	Fees	23
8.1	Rounding rule	23
9	Webhooks	24
9.1	Events	24
9.2	Payload	25
9.3	Verifying the signature	25
9.4	Anti-replay and deduplication	26
9.5	Retries	26
10	Errors	27
10.1	Common codes	28
11	Security	29
11.1	Key protection	29
11.2	Anti-replay	29
11.3	Test/live isolation	29
11.4	Checkout (iframe)	29
11.5	Webhooks	29
12	Sandbox (test data)	30
12.1	Charges	30
12.2	Payouts	30
12.3	Webhooks	30
12.4	Going live	31
13	API reference	32
13.1	Charges	32
13.2	Payouts	32
14	Limits and rate limiting	33

1 Introduction

bePago is a REST API for charging and paying in Paraguay. Two modules on the same integration:

- **Charges** — embed a checkout in an iframe and charge via QR SIP, Argentine QR or Pix. The charge is always created and credited in guaraníes. No sensitive data to handle.
- **Payouts** — send payouts to any bank account or wallet in the country through bank clearing. Built for payroll, suppliers or user payouts.

1.1 Base URL

live

```
https://bepago.com/api
```

All responses are JSON. The API currency is always PYG : amounts are integers in guaraníes, with no decimals.

1.2 Keys

When your account is created you receive two pairs of keys:

KEY	USE	WHERE IT GOES
pk_test_...	Test Key ID	X-API-Key header
sk_test_...	Test secret — HMAC signature	Server only, NEVER on the client
pk_live_...	Live Key ID	X-API-Key header
sk_live_...	Live secret — HMAC signature	Server only, NEVER on the client

Security. The `sk_` (secret) signs every request. If it leaks, anyone can operate as your merchant. Never put it in client code, in a public repository or in logs.

The **mode** (test or live) is always determined by the key you use. There is no parameter to force it. An `sk_test_` key only creates test resources.

1.3 Viewing the secret again

The **test** secret (`sk_test_`) can be viewed again in the panel whenever you need it (Integrations > API keys > *Show*): since it doesn't move real money, integration convenience comes first.

The **live** secret (`sk_live_`) is shown **once**, when the key is created. It cannot be viewed again: if you lose it, create another key, switch the secret on your server and revoke the old one.

2 Quickstart — first charge in 5 minutes

Secure integration checklist.

- **An order is completed only by a verified webhook or by querying the API** (GET `/v1/charges/:ref` with status `paid`). Never by the iframe's `postMessage` or by return URL parameters — those only drive the success screen.
- **The amount comes from your server, never from the browser**, and it is compared on confirmation: look up the order in your database, take the amount from there and check that it matches the charge before delivering.
- **The webhook verifies the signature and freshness, and discards repeated ids**: validate the HMAC signature, reject timestamps outside ± 5 min and store processed event ids so you never deliver twice.

Full example with the three rules done right: `examples/tienda-demo`.

With the official SDK the secure path is the only easy path: the amount comes from your server, the charge is idempotent per order and the SDK verifies the webhook. Four steps.

2.1 1. Install the SDK

Node (npm)

```
npm install bepago
```

PHP (Composer)

```
composer require bepago/sdk
```

No dependencies. Not using a package manager? Download the single file and drop it into your project: `bepago.mjs` (Node 18+) or `BePago.php` (PHP 7.4+).

2.2 2. A route that creates the charge with the amount from your database

The browser does NOT create the charge and does not know the amount: it asks your backend for a `checkout_url`. Your backend looks up the order in your database, uses `reference = orderId` (one charge per order) and the amount you have stored. If there is already an open charge for that order, you return the same `checkout_url` (idempotent).

Node — Express

```
import express from 'express';
import BePago from 'bepago';

const app = express();
const bp = BePago(process.env.BEPAGO_SECRET, { keyId: process.env.BEPAGO_KEY_ID });

app.post('/checkout/:orderId', async (req, res) => {
```

```

const order = await db.orders.find(req.params.orderId); // <- your database
if (!order) return res.status(404).json({ error: 'unknown order' });

// Idempotent: if there is already an open charge for this order, return the same
checkout_url.
if (order.checkoutUrl && order.status !== 'paid') {
  return res.json({ checkout_url: order.checkoutUrl });
}

const charge = await bp.charges.create({
  reference:    order.id,           // the reference IS the order id
  currency:     'PYG',
  amount:       order.amount,      // <- the amount comes from the server, ALWAYS
  description:  order.description,
  return_url:   `https://yourstore.com/thanks/${order.id}`,
});
await db.orders.update(order.id, { checkoutUrl: charge.checkout_url });
res.json({ checkout_url: charge.checkout_url });
});

```

PHP

```

use Bepago\BePago;

$bp = new BePago(getenv('BEPAGO_SECRET'), ['keyId' => getenv('BEPAGO_KEY_ID')]);

// POST /checkout/{orderId}
$order = $db->orders->find($orderId); // <- your database
if (!$order) { http_response_code(404); echo json_encode(['error' => 'unknown order']);
exit; }

// Idempotent: if there is already an open charge, return the same checkout_url.
if ($order->checkoutUrl && $order->status !== 'paid') {
  echo json_encode(['checkout_url' => $order->checkoutUrl]); exit;
}

$charge = $bp->charges()->create([
  'reference' => $order->id,           // the reference IS the order id
  'currency'  => 'PYG',
  'amount'    => $order->amount,      // <- the amount comes from the server, ALWAYS
  'description' => $order->description,
  'return_url' => "https://yourstore.com/thanks/{$order->id}",
]);
$db->orders->update($order->id, ['checkoutUrl' => $charge['checkout_url']]);
echo json_encode(['checkout_url' => $charge['checkout_url']]);

```

2.3 3. Embed the checkout with bepago.js

Load `bepago.js`, ask your backend for the `checkout_url` and embed it. `onComplete` is **UI and redirection only** (the buyer closed the checkout): your backend confirms the payment. That is why `onComplete` only gives you the `reference`, never a "paid" status.

HTML + bepago.js

```
<div id="bepago-checkout"></div>
<script src="https://bepago.com/sdk/bepago.js"></script>
<script>
  fetch('/checkout/' + orderId, { method: 'POST' })
    .then((r) => r.json())
    .then(({ checkout_url }) => {
      BePago.checkout({
        url: checkout_url,
        mount: '#bepago-checkout',
        // UI/redirection ONLY. Do NOT deliver here: this runs in the browser.
        onComplete: ({ reference }) => { location.href = '/thanks/' + reference; }
      });
    });
</script>
```

2.4 4. Receive the webhook with the SDK handler

The handler verifies the signature and freshness (± 5 min), discards duplicates by `x-bepago-event-id`, answers 200 and invokes `onPaid` **only** with valid events. Before delivering, you reconfirm through the API with `charges.confirm` and compare the amount against your order.

Node – Express

```
// The handler needs the RAW body: mount express.raw BEFORE the JSON parser.
const webhook = bp.webhooks.handler({
  secret: process.env.BEPAGO_WEBHOOK_SECRET,
  // store: in production, back it with your database (event ids table) so you
  // never deliver twice after a restart. The default is an in-memory Set.
  onPaid: async ({ reference }) => {
    const order = await db.orders.find(reference);
    const { paid, charge } = await bp.charges.confirm(reference); // source of truth
    if (paid && order && charge.amount === order.amount) await deliver(order);
  },
});

app.post('/webhooks/bepago', express.raw({ type: '*' }), webhook.express());
```

Node – Fastify

```
fastify.addContentTypeParser('*', { parseAs: 'string' }, (req, body, done) => done(null, body));
fastify.post('/webhooks/bepago', webhook.fastify());
```

PHP – PSR-7 (Slim, Laravel, Mezzio...)

```
use Psr\Http\Message\ServerRequestInterface as Req;
use Psr\Http\Message\ResponseInterface as Res;

$wh = $bp->webhooks()->handler([
  'secret' => getenv('BEPAGO_WEBHOOK_SECRET'),
  'store' => $myEventIds, // has()/add() against your database: never deliver twice
  'onPaid' => function (array $c) use ($bp, $db) {
    $order = $db->orders->find($c['reference']);
```

```

        $r = $bp->charges()->confirm($c['reference']); // reconfirm through the API
        if ($r['paid'] && $order && $r['charge']['amount'] === $order->amount)
        deliver($order);
    },
]);

$app->post('/webhooks/bepago', function (Req $req, Res $res) use ($wh) {
    $headers = [];
    foreach ($req->getHeaders() as $k => $v) $headers[$k] = $v[0] ?? '';
    $out = $wh->process($headers, (string) $req->getBody()); // RAW body
    $res->getBody()->write($out['body']);
    return $res->withStatus($out['status'])->withHeader('Content-Type',
    'application/json');
});

```

Note. This is **exactly** the code of the official example `examples/tienda-demo` : an end-to-end store with these four pieces, ready to run.

3 Official SDKs

The SDKs sign every request automatically. You don't build headers or compute HMAC yourself.

3.1 Node.js (18+, no dependencies)

Install with `npm install bePago`. Source code and issues: github.com/bepago/sdk-node. No npm? Download `bepago.mjs` and use `import BePago from './bepago.mjs'`.

```
import BePago from 'bepago'
```

```
import BePago from 'bepago';
const bp = BePago('sk_test_...', { keyId: 'pk_test_...' });

// Charges
const charge = await bp.charges.create({ reference: 'ord-1', currency: 'PYG', amount:
50000, description: 'Test' });
const found = await bp.charges.get('ord-1'); // by your reference
const { paid, charge: c } = await bp.charges.confirm('ord-1'); // verified status
(source of truth)
await bp.charges.refund('ord-1', 'Refund requested');
const bal = await bp.charges.balance(); // collected balance

// Payouts
const payout = await bp.payouts.create({
  amount: 1500000, currency: 'PYG',
  destination: { type: 'alias', alias: '0981600123' },
  description: 'Payroll',
}, 'unique-idem-key-per-payout');
const balance = await bp.payouts.balance();

// Batch (up to 1000 payouts): waits for approval in the panel unless auto-approval is on
const batch = await bp.payouts.batches.create({
  payouts: [{ amount: 500000, destination: { type: 'alias', alias: '0981600123' } },
external_id: 'pay-oct-001' }],
}, 'unique-idem-key-per-batch');
await bp.payouts.batches.get(batch.id);

// Webhooks: the handler verifies signature + freshness, discards duplicates and
// only invokes onPaid with valid events. express() and fastify() adapters.
const webhook = bp.webhooks.handler({ secret, onPaid: async ({ reference }) => { /*
deliver */ } });
app.post('/webhooks/bepago', express.raw({ type: '*/' })), webhook.express());
```

3.2 PHP (7.4+, no dependencies)

Install with `composer require bePago/sdk` (PSR-4, namespace `Bepago\BePago`). Source code and issues: github.com/bepago/sdk-php. No Composer? Download `BePago.php` (the same file as the package) and use `require_once 'BePago.php'; use Bepago\BePago;`.

```
composer require bepago/sdk
```

```
use Bepago\BePago;

$bp = new BePago('sk_test_...', ['keyId' => 'pk_test_...']);

$charge = $bp->charges()->create([
    'reference' => 'ord-1', 'currency' => 'PYG',
    'amount' => 50000, 'description' => 'Test'
]);
['paid' => $paid, 'charge' => $c] = $bp->charges()->confirm('ord-1'); // verified status

$payout = $bp->payouts()->create([
    'amount' => 1500000, 'currency' => 'PYG',
    'destination' => ['type' => 'alias', 'alias' => '0981600123'],
], 'unique-idem-key');

// Batch (up to 1000 payouts): waits for approval in the panel unless auto-approval is on
$batch = $bp->payouts()->batches()->create([
    'payouts' => [['amount' => 500000, 'destination' => ['type' => 'alias', 'alias' =>
    '0981600123'], 'external_id' => 'pay-oct-001']],
], 'unique-idem-key-per-batch');
$bp->payouts()->batches()->get($batch['id']);

// Webhooks: the handler verifies signature + freshness, discards duplicates and
// only invokes onPaid with valid events. process() returns status/body for PSR-7.
$wh = $bp->webhooks()->handler(['secret' => $secret, 'onPaid' => fn($c) => /* deliver */
null]);

// One-off verification of a SINGLE delivery (if you don't use the handler):
$evt = BePago::verifyWebhook($secret, ['id' => $id, 'timestamp' => $ts, 'signature' =>
$sig, 'body' => $body]);
```

3.3 Other languages

The HMAC v2 signature takes about 15 lines in any language. The [Authentication](#) section has copy-paste snippets in Node, PHP and Python with a test vector to validate your implementation.

3.4 Official example

The **demo store** is a minimal end-to-end integration (`node:http` + the SDK only): it creates a charge, embeds the checkout in an iframe, returns through `return_url` and receives the signed webhooks. Download the full server: `tienda-demo.mjs`.

4 Authentication

Where to set this up. Panel > Integrations > API keys.

Every API request is signed with HMAC-SHA256 (version 2). There are no Bearer tokens or OAuth: each request carries its own signature, computed with your secret key (`sk_`).

4.1 Required headers on every request

HEADER	VALUE	DESCRIPTION
X-API-Key	pk_test_...	Your Key ID (identifies the merchant)
X-Timestamp	1698000000000	Epoch in milliseconds. Window: ±5 min.
X-Nonce	a1b2c3d4e5f6...	Random, 16-128 chars [A-Za-z0-9_-]. A new one per request.
X-Signature-Version	2	Always 2.
X-Signature	cf24f5e5...	Hex HMAC-SHA256 of the canonical string.
X-Idempotency-Key	payout-001	Only on POST <code>/v1/payouts</code> (min 8 chars).

4.2 Canonical string

Format (one field per line)

```
v2
{timestamp}
{nonce}
{METHOD}
{path_with_query}
{idempotency_key_or_empty}
{sha256_hex_of_body}
```

If the request has no body (GET), the SHA-256 is that of the empty string: `e3b0c44...`.

Note. The `path` you sign is the API path (`/v1/...` , with its query string if any), **without** the `/api` prefix of the domain. You sign `/v1/charges` even though the request goes to `https://bepago.com/api/v1/charges`.

4.3 Implementation — Node

JavaScript (Node 18+)

```
import { createHash, createHmac, randomUUID } from 'node:crypto';

function signRequest(secret, method, path, body, idempotencyKey) {
  const timestamp = String(Date.now());
  const nonce = randomUUID().replaceAll('-', '');
  const bodyStr = body != null ? JSON.stringify(body) : '';
  const bodyHash = createHash('sha256').update(bodyStr, 'utf8').digest('hex');
  const canonical = ['v2', timestamp, nonce, method.toUpperCase(), path,
    idempotencyKey ?? '', bodyHash].join('\n');
  const signature = createHmac('sha256', secret).update(canonical, 'utf8').digest('hex');
  return { timestamp, nonce, signature };
}
```

4.4 Implementation — PHP

PHP

```
function signRequest(string $secret, string $method, string $path,
    ?string $body, ?string $idempotencyKey): array {
    $timestamp = (string)(int)(microtime(true) * 1000);
    $nonce = bin2hex(random_bytes(16));
    $bodyHash = hash('sha256', $body ?? '');
    $canonical = implode("\n", [
        'v2', $timestamp, $nonce, strtoupper($method),
        $path, $idempotencyKey ?? '', $bodyHash
    ]);
    $signature = hash_hmac('sha256', $canonical, $secret);
    return compact('timestamp', 'nonce', 'signature');
}
```

4.5 Implementation — Python

Python 3

```
import hashlib, hmac, time, uuid, json

def sign_request(secret, method, path, body=None, idempotency_key=''):
    timestamp = str(int(time.time() * 1000))
    nonce = uuid.uuid4().hex
    body_str = json.dumps(body, separators=(',', ':')) if body else ''
    body_hash = hashlib.sha256(body_str.encode()).hexdigest()
    canonical =
f"v2\n{timestamp}\n{nonce}\n{method.upper()}\n{path}\n{idempotency_key}\n{body_hash}"
    signature = hmac.new(secret.encode(), canonical.encode(), hashlib.sha256).hexdigest()
    return {'timestamp': timestamp, 'nonce': nonce, 'signature': signature}
```

Important. When sending the request you must send **the exact bytes you hashed**. With requests use `data=body_str` (the string you signed), **not** `json=body`: `json=` re-serializes the object (spaces, order) and the signature no longer matches.

4.6 Test vector

Use these fixed values to validate your implementation. If your function returns the same signature, it works:

Input

```
secret:      sk_test_demo_secret_key_1234567890
method:      POST
path:        /v1/charges
timestamp:    1698000000000
nonce:       abc123def456ghi789jk
idempotency_key: (empty)
body (JSON): {"reference":"pedido-001", "currency":"PYG", "amount":150000, "description":"Test"}
```

Expected signature

```
SHA-256 of body:  ac687e6ba5c15ff065b1850b4d5860d49bffe79a5fa3e368bb34044e1e52f42
X-Signature:       cf24f5e57bf986bf34801f49b9b1ee8b3ccf0dc39ced32fb8c10ba1c0082ea65
```

5 Charges

A charge represents a checkout for your customer to pay. You create the charge through the API, receive a URL, embed it in an iframe, and the payer chooses their payment method.

5.1 Create a charge

POST /v1/charges — scope: collect:write

FIELD	TYPE	REQUIRED	DESCRIPTION
reference	string	Yes	Your unique reference (order). Same reference = same charge (idempotent).
currency	string	Yes	Always PYG. Any other value returns 400 <code>validation_error</code> .
amount	integer	Yes	Amount in guaraníes, positive integer. The minimum is set by the payment method (QR SIP: Gs 1.000).
description	string	Yes	Description shown in the checkout (max 140 chars)
expires_in	integer	No	Checkout lifetime in seconds (60-604800, default 1800 = 30 min)
return_url	string	No	Post-payment return URL. Must be within your registered domains.
customer	object	No	name, email, document, phone
metadata	object	No	Arbitrary data. Stored with the charge; not included in the webhook or the response.

Note. The `reference` is idempotent: if you send the same reference twice, you receive the same charge (200 instead of 201). Retries never create duplicate charges.

5.2 Retrieve a charge

GET /v1/charges/:ref — scope: collect:read

You can use your `reference` or the bePago `id`.

5.3 List charges

GET /v1/charges — scope: collect:read

Returns your charges from newest to oldest, paginated by cursor. Optional query parameters: `limit` (1 to 100, default 20), `status` (filters by status), `from` and `to` (range over `created_at`, ISO 8601 or YYYY-MM-DD) and `cursor` (the `next_cursor` from the previous page).

Response

```
{
  "object": "list",
  "livemode": false,
  "data": [ /* charges, same format as GET /v1/charges/:ref */ ],
  "has_more": true,
  "next_cursor": "chg_..."
}
```

To fetch the next page, repeat the request with `cursor=<next_cursor>`. When `has_more` is `false`, there are no more results.

5.4 Refund

POST `/v1/charges/:ref/refund` — scope: `collect:write`

The `reason` field is **required** (3 to 200 characters): it is recorded in the refund's audit trail.

Body

```
{ "reason": "Customer requested a refund" }
```

5.5 Cancel

POST `/v1/charges/:ref/cancel` — scope: `collect:write`

Cancels a charge that has **not been paid yet** (`created` or `pending`): the checkout expires and nothing is credited. A paid charge is not cancelled, it is refunded (returns `409`). No body.

5.6 Collected balance

GET `/v1/charges/balance` — scope: `collect:read`

Returns the available balance and the fees pending collection, in guaraníes.

Response

```
{
  "livemode": false,
  "balances": [
    { "currency": "PYG", "available": 4500000, "fees_payable": 112500 }
  ]
}
```

5.7 Charge statuses

STATUS	MEANING	FINAL
created	Charge created, checkout active. The payer has not done anything yet.	No
pending	The payer started the payment (QR generated, awaiting confirmation).	No
paid	Payment confirmed. You can deliver the product.	No (moves to settled, or to refunded if you refund it)
settled	The money is available for withdrawal.	No (can still move to refunded)
expired	The checkout expired without payment.	Yes
refunded	Refunded.	Yes
failed	The charge failed.	Yes

6 Embedded checkout

Where to set this up. Panel > Integrations > Checkout domain.

The bePago checkout is an iframe you load on your site. The payer chooses the payment method; you never touch sensitive data.

Checklist. Before embedding the checkout, go through the [secure integration checklist](#) in the Quickstart: an order is completed only by a verified webhook or by the API, the amount comes from your server, and the webhook is verified and deduplicated.

6.1 Embed with bepago.js (recommended)

The `bepago.js` script renders the iframe and tells you when the buyer finished the screen. Ask your backend for the `checkout_url` (your backend creates the charge with the server-side amount) and embed it.

HTML + bepago.js

```
<div id="bepago-checkout"></div>
<script src="https://bepago.com/sdk/bepago.js"></script>
<script>
  BePago.checkout({
    url: checkout_url,          // the checkout_url returned by YOUR backend
    mount: '#bepago-checkout',
    onComplete: ({ reference }) => {
      // UI and redirection only; confirmation belongs to the backend
      location.href = '/thanks/' + reference;
    }
  });
</script>
```

Security. `onComplete` means "the buyer closed the checkout", **not** "the payment is confirmed". That is why you only receive `reference` and no status: your server decides whether to deliver, through a webhook verified with the SDK or with `bp.charges.confirm(reference)`. Never deliver from the browser.

6.2 Allowed domains

For security, the iframe can only be loaded from the domains you registered under **Integrations > Checkout domain** in your panel. The `frame-ancestors` header restricts it to those exact origins.

6.3 Payment methods

The charge is created in guaraníes. bePago enables the methods available to each merchant; the checkout shows only the enabled ones.

METHOD	NETWORK	PAYER CURRENCY	YOU RECEIVE
QR SIP	SIP · BCP	Guaraníes (PYG)	PYG
QR AR	Interoperable QR	Argentine pesos (ARS)	PYG
Pix	Pix (BCB)	Reais (BRL)	PYG

The payer pays in the currency of their method; you always receive guaraníes, for the amount of the charge.

7 Payouts

Send payouts to any bank account or wallet in Paraguay through bank clearing.

7.1 Create a payout

POST /v1/payouts — scope: payouts:write

Important. Every payout requires a unique `X-Idempotency-Key` header (min 8 chars). If you resend the same request with the same key, you receive the same payout (`200` instead of `201`) without creating a new one.

7.2 Bank account destination

body

```
{
  "amount": 3200000,
  "currency": "PYG",
  "destination": {
    "type": "bank_account",
    "bank_code": "0017",
    "account_number": "0012345678",
    "document": "4567890",
    "holder_name": "Maria Cabrera"
  },
  "description": "October payroll",
  "external_id": "pay-oct-001"
}
```

7.3 Alias destination

body

```
{
  "amount": 5000000,
  "currency": "PYG",
  "destination": {
    "type": "alias",
    "alias": "0981600123"
  },
  "description": "Supplier payment"
}
```

The alias can be a phone number, national ID or email. bePago resolves it against the SIPAP directory and confirms the account holder's name.

7.4 Retrieve

GET /v1/payouts/:id — scope: payouts:read

7.5 List payouts

GET /v1/payouts — scope: payouts:read

Returns your payouts from newest to oldest, paginated by cursor. Same query parameters as the charges list: `limit` (1 to 100, default 20), `status`, `from`, `to` and `cursor`. The response carries `data`, `has_more` and `next_cursor` with the same pagination mechanics.

7.6 Cancel

POST /v1/payouts/:id/cancel — scope: payouts:write

A payout can only be cancelled if it has not been sent to the bank yet (`ready` or `held`).

7.7 Balance

GET /v1/balance — scope: balance:read

Response

```
{
  "available": 45000000,
  "reserved": 3200000,
  "fees_payable": 15000,
  "currency": "PYG",
  "livemode": false
}
```

7.8 Statuses

STATUS	MEANING	FINAL
created	Just created	No
validating	Validating the recipient's data	No
ready	Queued, goes to the bank through clearing	No
held	Held: a supervisor must approve it	No
submitting	Sending to the bank	No
submitted	The bank is processing it	No
unknown	No confirmation from the bank. Platform verifies it. Do NOT retry.	No
completed	Credited to the recipient's account	Yes
failed	Failed: the money returned to the balance	Yes
rejected	Rejected by a supervisor	Yes
cancelled	Cancelled before going to the bank	Yes
expired	Nobody approved it in time	Yes

Security. If a payout ends up in `unknown`, **do NOT retry it**. Platform verifies it manually. Retrying can pay twice.

7.9 Batches (bulk payouts)

To pay many destinations at once there are two paths: the **API** (your system uploads and runs the batch in one call) or the **panel** (you upload a CSV, review it and another user approves it — four eyes).

Through the API

POST `/v1/payouts/batches` — scope: `payouts:write`. With the SDK:
`bp.payouts.batches.create(body, idempotencyKey)` (Node) or `$bp->payouts()->batches()->create($body, $idempotencyKey)` (PHP); the status with `batches.get(id)`.

Important. The whole batch carries one unique `X-Idempotency-Key` header (min 8 chars). On top of that, each payout in the batch is idempotent by its position (`batch:<id>:<row>`): if you resend the same batch, **nothing is ever paid twice**. The balance is validated for the **entire** batch before any payout is created.

Note. Four eyes through the API: by default, a batch uploaded through the API stays `pending_approval` and needs a panel user to approve it (different from whoever holds the key). To enable auto-approval, ask support to turn the option on for your merchant; the key also needs the `payouts:approve` scope and **at least one IP in its allowlist**. Without all three conditions, the batch always waits for manual approval.

body

```
{
  "filename": "payroll-oct.json",
  "payouts": [
    {
      "amount": 500000,
      "destination": { "type": "alias", "alias": "0981600123" },
      "external_id": "pay-oct-001",
      "description": "October payroll"
    },
    {
      "amount": 3200000,
      "destination": {
        "type": "bank_account",
        "bank_code": "0017",
        "account_number": "0012345678",
        "document": "4567890",
        "holder_name": "Maria Cabrera"
      },
      "external_id": "pay-oct-002"
    }
  ]
}
```

Response

```
{
  "id": "btc...",
  "status": "pending_approval",
  "livemode": false,
  "total_rows": 2,
  "valid_rows": 2,
  "invalid_rows": 0,
  "total_amount": 3700000,
  "total_fee": 44400,
  "currency": "PYG",
  "invalid": [],
  "payouts": [
    { "row": 0, "id": "pay...", "status": "ready", "amount": 500000, "fee": 6000, "net": 506000, "reference": "pay-oct-001", "operation_number": null, "failure_code": null, "beneficiary_name": null },
    { "row": 1, "id": "pay...", "status": "ready", "amount": 3200000, "fee": 38400, "net": 3238400, "reference": "pay-oct-002", "operation_number": null, "failure_code": null, "beneficiary_name": "Maria Cabrera" }
  ]
}
```

A batch starts as `pending_approval` and moves to `processing` once approved (manually or automatically). When every payout reaches a final status it becomes `completed`, or `completed_with_errors` if any row failed; a batch rejected before running becomes `cancelled`. Each payout follows the same [statuses](#) as a single payout.

`GET /v1/payouts/batches/:id` — scope: `payouts:read` — returns the same body with the updated status of each row (operation number and confirmed account holder once credited). The fee amounts in the example are illustrative: `fee` and `net` come from your contract rate.

Note. If one payout in the batch fails, the others continue and the failed amount returns to the balance with its own ledger entry. A batch is never partially paid for lack of balance: either it covers everything, or nothing is created.

Through the panel (CSV)

Under **Batches (CSV)** in the panel you upload a file, see the preview with valid and invalid rows (and the reason for each), the total amount and what it deducts from the balance. Nothing is paid until **another user** approves it (whoever uploaded cannot approve their own batch). When it finishes you can export the result as CSV with the status, operation number and confirmed account holder of each payout.

The CSV has a header row with these columns (a template is available for download on the same screen):

COLUMN	PURPOSE	REQUIRED
tipo	alias or cuenta	Yes
alias	Phone, national ID or email (if tipo=alias)	If alias
banco	SIPAP bank code (if tipo=cuenta)	If account
cuenta	Account number	If account
documento	Account holder's national ID or RUC	If account
titular	Account holder's name	If account
monto	Amount in guaraníes (integer)	Yes
referencia	Your identifier for the row	No
descripcion	Free text	No

CSV example

```
tipo,alias,banco,cuenta,documento,titular,monto,referencia,descripcion
alias,0981600123,,,,,500000,pay-oct-001,October payroll
cuenta,,0017,0012345678,4567890,Maria Cabrera,3200000,pay-oct-002,October payroll
```

Note. In **test mode** the whole batch flow works the same against the sandbox: payouts never reach a bank. An alias or account containing **invalido** (or an account starting with **9999**) makes that row fail on purpose, to test the partial-failure path. See [Sandbox](#).

8 Fees

Your merchant fee is the one agreed in your contract with bePago, and you see it applied in your panel: on every charge and payout, in the day's balance and in the statements. Do not compute it in your system: every charge and every payout return the fee and net fields already calculated, and the webhook includes them. This section explains how they are applied and rounded so your reconciliation balances to the guaraní.

On a charge, `fee` is the fee and `net` what is credited to your balance (`amount - fee`); `expected_net` is the net expected when it was created: the charge rate depends on the payment method (QR SIP, Pix, QR Argentina), so until the payer picks one it is estimated with the highest-fee method, and it is fixed once chosen. On a payout, `fee` is the fee and `net` what leaves your balance (`amount + fee`). Batches carry `fee` and `net` per row and the batch `total_fee`. The `charge.*` and `payout.*` webhooks carry the same fields as the API.

The fee is charged at one of two moments, depending on how your merchant is configured:

MODE	WHEN IT IS CHARGED	HOW IT IS CALCULATED
<code>per_transaction</code>	On each payout	An optional fixed amount plus a percentage of the payout amount
<code>on_funding</code>	When a top-up is credited	A percentage of the deposit (no fixed amount). Payouts go out without a fee.

The percentage is expressed in **basis points** (bps): 1 bps is one hundredth of a percentage point, 10,000 bps is 100 %. The fee is accrued with its own ledger entry at the moment of the movement, so what you see on the payout is exactly what is recorded.

8.1 Rounding rule

Guaraníes have no decimals. When the percentage yields a fraction, the fee is **always rounded up** to the whole guaraní. One rule, identical for payouts and charges:

fee rounding (up to the whole guaraní), with `T` = your contract rate in basis points

```
fee = ceil(amount * T / 10000) + fixed
# on charges, if your contract has a minimum: fee = max(fee, minimum)
```

When the calculation yields a fraction it goes up to the next integer; when it yields an exact integer, nothing is added. One sample row:

AMOUNT	RATE	AMOUNT × T / 10000	FEE
Gs 155.555	T = 137 bps (example rate, not your fee)	2.131,1035	Gs 2.132

Note. Rounding always goes to the next whole amount: the fee never ends in fractions of a guaraní and the ledger entry balances to the guaraní.

9 Webhooks

Where to set this up. Panel > Integrations > Webhook.

bePago sends signed webhooks to your endpoint when a resource changes status. You configure them under **Integrations > Webhook** in your panel.

9.1 Events

EVENT	MODULE	WHEN IT FIRES
charge.paid	Charges	The charge was paid
charge.settled	Charges	The money is available for withdrawal
charge.expired	Charges	The checkout expired without payment
charge.refunded	Charges	Refunded
charge.failed	Charges	Failed
payout.processing	Payouts	The payout is being processed
payout.completed	Payouts	Completed
payout.failed	Payouts	Failed
payout.held	Payouts	Held (needs approval)
payout.cancelled	Payouts	Cancelled
payout.expired	Payouts	Held with no decision in time: it expired and the amount returned to the balance
payout.unknown	Payouts	No confirmation from the bank

9.2 Payload

charge.paid example

```
{
  "id": "clx8y2z0001...",
  "event": "charge.paid",
  "livemode": true,
  "merchant_id": "1042",
  "data": {
    "id": "chg_abc123",
    "reference": "order-001",
    "status": "paid",
    "amount": 150000,
    "fee": 3750,
    "net": 146250,
    "expected_net": 146250,
    "currency": "PYG",
    "method": "qr_sip",
    "paid_at": "2026-10-22T09:05:00.000Z"
  }
}
```

`merchant_id` is your merchant ID, a number (the same one shown in the panel as “ID de comercio: 1042”). The amounts in the example are illustrative: `fee` and `net` come from your contract rate. The webhook carries the same ledger `net` as the API and the panel: on `charge.paid` it matches `expected_net`; on `charge.refunded` it goes down (the refund leaves the balance) and on `charge.expired` it is `0`.

9.3 Verifying the signature

Every webhook carries `X-Webhook-Timestamp`, `X-Webhook-Signature` and `X-BePago-Event-Id`. The signature is `HMAC-SHA256(webhook_secret, timestamp + body)`. Verify **both the signature and the timestamp freshness** (reject deliveries outside ± 5 minutes): a captured and replayed old webhook never gets through.

Node

```
import { createHmac, timingSafeEqual } from 'node:crypto';

function verifyWebhook(secret, timestamp, body, signature) {
  if (Math.abs(Date.now() - Number(timestamp)) > 5 * 60 * 1000) return false; //
  freshness  $\pm 5$  min
  const expected = createHmac('sha256', secret)
    .update(timestamp + body, 'utf8').digest('hex');
  const a = Buffer.from(expected, 'hex');
  const b = Buffer.from(signature, 'hex');
  return a.length === b.length && timingSafeEqual(a, b);
}
```

PHP

```
function verifyWebhook(string $secret, string $ts, string $body, string $sig): bool {
    if (abs((int)(microtime(true) * 1000) - (int)$ts) > 5 * 60 * 1000) return false; //
    freshness ±5 min
    return hash_equals(hash_hmac('sha256', $ts . $body, $secret), $sig);
}
```

Python

```
import hmac, hashlib, time
def verify_webhook(secret, timestamp, body, signature):
    if abs(int(time.time() * 1000) - int(timestamp)) > 5 * 60 * 1000: # freshness ±5 min
        return False
    expected = hmac.new(secret.encode(), (timestamp + body).encode(),
        hashlib.sha256).hexdigest()
    return hmac.compare_digest(expected, signature)
```

9.4 Anti-replay and deduplication

Every event carries a unique `id` (in the body and in the `X-BePago-Event-Id` header). Retries resend the **same id** with a fresh timestamp, so the signature alone cannot tell a legitimate retry from a duplicate. Store the processed ids and discard repeats so your processing is idempotent.

Node

```
const eventId = req.headers['x-bepago-event-id'];
if (await alreadyProcessed(eventId)) return res.sendStatus(200); // duplicate
await markProcessed(eventId);
// ... process the event once
```

9.5 Retries

If your endpoint does not answer 2xx, bePago retries with increasing backoff (1min, 5min, 15min, 1h, 6h, 12h, 24h, 48h, 72h): 9 attempts over up to 3 days, to tolerate long outages of your merchant's server. After the last failed attempt, the delivery is marked as failed and stays in the panel for manual resend. The event `id` is stable across retries.

Note. A lost webhook breaks nothing. You can always query the status through the API. Webhooks are a convenience, not the source of truth.

10 Errors

All error responses follow the same structure:

error format

```
{
  "error": {
    "code": "insufficient_funds",
    "message": "Saldo insuficiente para este pago.",
    "action": "contact_platform",
    "request_id": "req_abc123"
  }
}
```

The `action` field tells you what to do: `retry`, `fix_input`, `reauthenticate`, `contact_admin`, `contact_platform`, `wait` or `none`.

10.1 Common codes

CODE	HTTP	WHAT TO DO
<code>unauthorized</code>	401	Authentication failure. <code>details.reason</code> says which: <code>bad_signature</code> (check the canonical string), <code>expired</code> (timestamp outside ± 5 min, sync your clock), <code>replayed_nonce</code> (generate a new nonce), <code>ip_not_allowed</code> , <code>key_revoked</code> , <code>unknown_key</code> .
<code>signature_version_unsupported</code>	401	Send <code>X-Signature-Version: 2</code>
<code>insufficient_scope</code>	403	The key lacks the required scope (<code>details.required_scope</code>). Create a key that includes it.
<code>module_not_enabled</code>	403	Your merchant does not have that module enabled
<code>validation_error</code>	400	Check the body fields
<code>insufficient_funds</code>	422	No balance. Contact platform to top up.
<code>limit_exceeded</code>	422	The payout exceeds a limit configured for your merchant
<code>missing_idempotency_key</code>	400	Add <code>X-Idempotency-Key</code> on payout and batch POSTs (min 8 chars)
<code>idempotency_conflict</code>	409	That idempotency key was already used with different content
<code>rate_limited</code>	429	Wait and retry
<code>not_found</code>	404	The resource does not exist
<code>not_cancellable</code>	409	The payout already went to the bank or is closed
<code>charge_not_refundable</code>	409	The charge cannot be refunded in its current status
<code>checkout_closed</code>	409	The charge no longer accepts payment or cancellation (paid, expired or cancelled)
<code>checkout_domain_invalid</code>	422	Use the exact https origin when registering the checkout domain
<code>duplicate_suspect</code>	409	Possible duplicate payout (same destination and amount recently)

11 Security

11.1 Key protection

- The `sk_` (secret key) NEVER goes on the client (browser, mobile app, iframe). Server only.
- The `pk_` (Key ID) identifies your merchant but cannot sign. It can be exposed.
- If a secret leaked or you lost it: create a new key, switch it on your server and revoke the old one from the panel. A revoked key stops authenticating immediately.
- IP allowlist: optional, but recommended in live mode. With the list set, a request from any other IP is rejected even with a valid signature.

11.2 Anti-replay

- Timestamp window: ± 5 minutes. Sync your server clock with NTP.
- The nonce is consumed on first use. A captured signature cannot be replayed.
- The idempotency key is signed: it cannot be changed to duplicate a payout.

11.3 Test/live isolation

- Test and live keys live in completely separate universes.
- A test payout can NEVER count as real.
- The `livemode` field in every response tells whether the resource is real or test.
- The mode always comes from the key, never from the body.

11.4 Checkout (iframe)

- `frame-ancestors` restricted to your registered domains.
- Never pass sensitive data in the `checkout_url` query string.
- Validate the `origin` of `postMessage` events before processing them.

11.5 Webhooks

- Always verify the HMAC signature before processing a webhook.
- Always confirm the status through the API after receiving a webhook. The webhook is a notification, not the source of truth.
- Use `timingSafeEqual` (Node) or `hash_equals` (PHP) to compare signatures (prevents timing attacks).

12 Sandbox (test data)

Where to set this up. Panel > Integrations > Test charge.

With your `sk_test_` keys you can test the whole integration without moving real money.

12.1 Charges

First you need a test charge. Create it through the API (`POST /v1/charges` with your `sk_test_` key) or, without writing code, from the panel under **Integrations > Test charge**.

With a test key there are two equivalent ways to simulate the payment of a charge:

1. **From the test checkout:** open the charge URL and choose a payment method. The test checkout charges nothing real — choosing the method marks the charge as paid, just like in live mode.
2. **Through the API:** with the sandbox endpoint, without opening the checkout.

simulate payment through the API (`sk_test` only)

```
POST /v1/test/charges/{reference}/pay
```

Either one moves the charge to `paid`, creates the ledger entries (the balance goes up) and fires the `charge.paid` webhook. This lets you test the full cycle without moving real money.

Example

```
const charge = await bp.charges.create({ reference: 'test-1', currency: 'PYG', amount:
50000, description: 'Test' });
// Simulate the payment
await fetch(BASE + '/v1/test/charges/test-1/pay', { method: 'POST', headers: {
...hmacHeaders } });
// The charge is now 'paid' and the balance went up
```

A charge whose checkout is never opened stays in `created`: creating the charge only generates the payment URL, it moves no money. As long as nobody opens the checkout and you don't call the sandbox endpoint, the charge stays in `created` (until it expires). Choosing the payment method moves it to `pending` (QR issued) and, once confirmed, to `paid`.

12.2 Payouts

Test payouts simulate the full cycle (validation, submission, result) without touching any bank. A destination with `invalido` in the alias, or an account starting with `9999`, ends in `failed` on purpose; everything else is credited.

12.3 Webhooks

They are sent the same way in test and live. The `livemode: false` field tells them apart. Your endpoint receives them identically.

12.4 Going live

1. Switch your keys from `sk_test_` to `sk_live_`.
2. Set the IP allowlist on your live key (optional, recommended).
3. Register your checkout domains.
4. The same code works: there is nothing to rewrite.

13 API reference

13.1 Charges

METHOD	ENDPOINT	SCOPE	DESCRIPTION
POST	/v1/charges	collect:write	Create a charge (idempotent by reference)
GET	/v1/charges/:ref	collect:read	Retrieve (by reference or ID)
GET	/v1/charges	collect:read	List (cursor + status/date filters)
POST	/v1/charges/:ref/refund	collect:write	Refund
POST	/v1/charges/:ref/cancel	collect:write	Cancel (created/pending only)
GET	/v1/charges/balance	collect:read	Collected balance and pending fees
POST	/v1/test/charges/:ref/pay	collect:write	Simulate the payment (sk_test only)

13.2 Payouts

METHOD	ENDPOINT	SCOPE	DESCRIPTION
POST	/v1/payouts	payouts:write	Create a payout (requires X-Idempotency-Key)
GET	/v1/payouts/:id	payouts:read	Retrieve a payout
GET	/v1/payouts	payouts:read	List (cursor + status/date filters)
POST	/v1/payouts/:id/cancel	payouts:write	Cancel (ready/held only)
POST	/v1/payouts/batches	payouts:write	Payout batch (requires X-Idempotency-Key)
GET	/v1/payouts/batches/:id	payouts:read	Retrieve a batch
GET	/v1/balance	balance:read	Available and reserved balance

14 Limits and rate limiting

LIMIT	VALUE
Requests per minute (per key)	600
Requests per minute (per IP)	300
Timestamp window	±5 minutes
Nonce	16-128 chars, single use
Maximum amount per request	Gs 1.000.000.000.000
Payouts per batch	1,000
Idempotency key	Min 8 chars
Maximum body	512 KB

If you exceed the rate limit you receive `429 rate_limited`. Wait and retry with exponential backoff.

Need help with the integration? Write to hola@bepago.com