bePago

DOCUMENTACIÓN PARA DESARROLLADORES

Guía de integración

API de cobros por QR y pagos masivos en Paraguay

Contenido

1 **Introducción** 3

1.1 Base URL 3
1.2 Claves 3
1.3 Ver el secreto otra vez 3

2 **Quickstart — primer cobro en 5 minutos** 4

2.1 1. Instalá el SDK 4
2.2 2. Ruta que crea el cobro con el monto de tu base 4
2.3 3. Embebé el checkout con bepago.js 5
2.4 4. Recibí el webhook con el handler del SDK 6

3 **SDKs oficiales** 8

3.1 Node.js (18+, sin dependencias) 8
3.2 PHP (7.4+, sin dependencias) 8
3.3 Otros lenguajes 9
3.4 Ejemplo oficial 9

4 **Autenticación** 10

4.1 Headers requeridos en cada request 10
4.2 Cadena canónica 10
4.3 Implementación — Node 11
4.4 Implementación — PHP 11
4.5 Implementación — Python 11
4.6 Vector de prueba 12

5 **Cobros** 13

5.1 Crear un cobro 13
5.2 Consultar un cobro 13
5.3 Listar cobros 13
5.4 Reembolsar 14
5.5 Cancelar 14
5.6 Saldo cobrado 14
5.7 Estados de un cobro 15

6 **Checkout embebido** 16

6.1 Embeber con bepago.js (recomendado) 16
6.2 Dominios permitidos 16
6.3 Medios de pago 16

7 **Pagos Masivos** 18

7.1 Crear un pago 18
7.2 Destino por cuenta bancaria 18
7.3 Destino por alias 18
7.4 Consultar 18
7.5 Listar pagos 19
7.6 Cancelar 19
7.7 Saldo 19
7.8 Estados 19
7.9 Lotes (dispersión masiva) 20

8 **Comisiones** 23
8.1 Regla de redondeo 23

9 **Webhooks** 24
9.1 Eventos 24
9.2 Payload 25
9.3 Verificar la firma 25
9.4 Anti-replay y deduplicación 26
9.5 Reintentos 26

10 **Errores** 27
10.1 Códigos comunes 28

11 **Seguridad** 29
11.1 Protección de claves 29
11.2 Anti-replay 29
11.3 Aislamiento test/live 29
11.4 Checkout (iframe) 29
11.5 Webhooks 29

12 **Sandbox (datos de prueba)** 30
12.1 Cobros 30
12.2 Pagos masivos 30
12.3 Webhooks 30
12.4 Pasar a producción 31

13 **Referencia de la API** 32
13.1 Cobros 32
13.2 Pagos Masivos 32

14 **Límites y rate limiting** 33

1 Introducción

bePago es una API REST para cobrar y pagar en Paraguay. Dos módulos sobre la misma integración:

- **Cobros** — embebés un checkout por iframe y cobrás por QR SIP, QR argentino o Pix. El cobro se crea y se acredita siempre en guaraníes. Sin manejar datos sensibles.
- **Pagos Masivos** — dispersás pagos a cualquier cuenta bancaria o billetera del país por clearing bancario. Ideal para nómina, proveedores o pagos a usuarios.

1.1 Base URL

Producción

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

Todas las respuestas son JSON. La moneda de la API es siempre `PYG` : los montos son enteros en guaraníes, sin decimales.

1.2 Claves

Al crear tu cuenta recibís dos pares de claves:

CLAVE	USO	DONDE VA
`pk_test_…`	Key ID de prueba	Header `X-Api-Key`
`sk_test_…`	Secret de prueba — firma HMAC	Solo en tu server, NUNCA en el cliente
`pk_live_…`	Key ID de producción	Header `X-Api-Key`
`sk_live_…`	Secret de producción — firma HMAC	Solo en tu server, NUNCA en el cliente

Seguridad. El `sk_` (secret) firma cada request. Si se filtra, cualquiera puede operar como tu comercio. Nunca lo pongas en código del cliente, en un repositorio público ni en logs.

El **modo** (prueba o producción) se determina siempre por la clave que usás. No hay un parámetro para forzarlo. Una clave `sk_test_` solo crea recursos de prueba.

1.3 Ver el secreto otra vez

El secreto de **prueba** (`sk_test_`) se puede volver a ver en el panel cuando lo necesites (Integraciones › Claves de API › *Ver*): como no mueve plata real, se prioriza la comodidad de integrar.

El secreto de **producción** (`sk_live_`) se muestra **una sola vez**, al crear la key. No se puede volver a ver: si lo perdés, creá otra key, cambiá el secreto en tu servidor y revocá la anterior.

2 Quickstart — primer cobro en 5 minutos

Checklist de integración segura.

- **La orden se completa solo por webhook verificado o consultando la API** (GET /v1/charges/:ref con status `paid`). Nunca por el `postMessage` del iframe ni por parámetros de la URL de retorno — eso solo sirve para mostrar la pantalla de éxito.
- **El monto sale de tu servidor, nunca del navegador**, y se compara al confirmar: buscá la orden en tu base, tomá el importe de ahí y verificá que coincida con el del cobro antes de entregar.
- **El webhook verifica firma y frescura, y descarta ids repetidos**: validá la firma HMAC, rechazá timestamps fuera de ±5 min y guardá los event ids ya procesados para no entregar dos veces.

Ejemplo completo con las tres reglas bien hechas: `examples/tienda-demo`.

Con el SDK oficial el camino seguro es el único camino fácil: el monto sale de tu servidor, el cobro es idempotente por orden y el webhook lo verifica el SDK. Cuatro pasos.

2.1 1. Instalá el SDK

Node (npm)

```
npm install bepago
```

PHP (Composer)

```
composer require bepago/sdk
```

Sin dependencias. ¿No usás gestor de paquetes? Bajá el archivo único y dejalo en tu proyecto: `bepago.mjs` (Node 18+) o `BePago.php` (PHP 7.4+).

2.2 2. Ruta que crea el cobro con el monto de tu base

El navegador NO crea el cobro ni conoce el monto: le pide un `checkout_url` a tu backend. Tu backend busca la orden en tu base, usa `reference = orderId` (un solo cobro por orden) y el importe que tenés guardado. Si ya hay un cobro abierto para esa orden, devolvés el mismo `checkout_url` (idempotente).

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);   // <- tu base de datos
  if (!order) return res.status(404).json({ error: 'orden desconocida' });
```

```
  // Idempotente: si ya hay un cobro abierto para esta orden, el mismo checkout_url.
  if (order.checkoutUrl && order.status !== 'paid') {
    return res.json({ checkout_url: order.checkoutUrl });
  }

  const charge = await bp.charges.create({
    reference:   order.id,            // la referencia ES el id de la orden
    currency:    'PYG',
    amount:      order.amount,        // <- el importe sale del servidor, SIEMPRE
    description: order.description,
    return_url:  `https://tutienda.com/gracias/${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);                 // <- tu base de datos
if (!$order) { http_response_code(404); echo json_encode(['error' => 'orden
desconocida']); exit; }

// Idempotente: si ya hay un cobro abierto, el mismo checkout_url.
if ($order->checkoutUrl && $order->status !== 'paid') {
    echo json_encode(['checkout_url' => $order->checkoutUrl]); exit;
}

$charge = $bp->charges()->create([
    'reference'   => $order->id,          // la referencia ES el id de la orden
    'currency'    => 'PYG',
    'amount'      => $order->amount,      // <- el importe sale del servidor, SIEMPRE
    'description' => $order->description,
    'return_url'  => "https://tutienda.com/gracias/{$order->id}",
]);
$db->orders->update($order->id, ['checkoutUrl' => $charge['checkout_url']]);
echo json_encode(['checkout_url' => $charge['checkout_url']]);
```

2.3 3. Embebé el checkout con bepago.js

Cargá `bepago.js`, pedí el `checkout_url` a tu backend y embebelo. `onComplete` es **solo UI y redirección** (el comprador cerró el checkout): la confirmación del pago la hace tu backend. Por eso `onComplete` solo te da la `reference`, nunca un estado tipo “pagado”.

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',
        // SOLO UI/redirección. NO entregues acá: esto corre en el navegador.
        onComplete: ({ reference }) => { location.href = '/gracias/' + reference; }
      });
    });
</script>
```

2.4 4. Recibí el webhook con el handler del SDK

El handler verifica firma y frescura (±5 min), descarta duplicados por `x-bepago-event-id`, responde 200 y **solo** invoca `onPaid` con eventos válidos. Antes de entregar, reconfirmás por API con `charges.confirm` y comparás el monto contra tu orden.

Node — Express

```
// El handler necesita el cuerpo CRUDO: montá express.raw ANTES del parser JSON.
const webhook = bp.webhooks.handler({
  secret: process.env.BEPAGO_WEBHOOK_SECRET,
  // store: en producción, respaldalo en tu base (tabla de event ids) para no
  // entregar dos veces tras un reinicio. Por defecto es un Set en memoria.
  onPaid: async ({ reference }) => {
    const order = await db.orders.find(reference);
    const { paid, charge } = await bp.charges.confirm(reference);   // fuente de verdad
    if (paid && order && charge.amount === order.amount) await entregar(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'  => $misEventIds,   // has()/add() contra tu base: no entregar dos veces
    'onPaid' => function (array $c) use ($bp, $db) {
        $order = $db->orders->find($c['reference']);
```

```
        $r = $bp->charges()->confirm($c['reference']);   // reconfirmamos por API
        if ($r['paid'] && $order && $r['charge']['amount'] === $order->amount)
entregar($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());   // cuerpo CRUDO
    $res->getBody()->write($out['body']);
    return $res->withStatus($out['status'])->withHeader('Content-Type',
'application/json');
});
```

Nota. Este es **exactamente** el código del ejemplo oficial `examples/tienda-demo` : una tienda end-to-end con estas cuatro piezas, lista para correr.

3 SDKs oficiales

Los SDKs firman cada request automáticamente. No tenés que armar headers ni calcular HMAC.

3.1 Node.js (18+, sin dependencias)

Instalá con `npm install bepago`. Código fuente y issues: github.com/bepago/sdk-node. ¿Sin npm? Bajá `bepago.mjs` y usá `import BePago from './bepago.mjs'`.

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

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

// Cobros
const charge = await bp.charges.create({ reference: 'ord-1', currency: 'PYG', amount:
50000, description: 'Test' });
const found  = await bp.charges.get('ord-1');           // por tu referencia
const { paid, charge: c } = await bp.charges.confirm('ord-1');  // estado verificado
(fuente de verdad)
await bp.charges.refund('ord-1', 'Devolucion solicitada');
const bal    = await bp.charges.balance();               // saldo cobrado

// Pagos masivos
const payout = await bp.payouts.create({
  amount: 1500000, currency: 'PYG',
  destination: { type: 'alias', alias: '0981600123' },
  description: 'Quincena',
}, 'idem-key-unica-por-pago');
const saldo  = await bp.payouts.balance();

// Lote (hasta 1000 pagos): queda pendiente de aprobación en el panel salvo auto-
aprobación
const lote = await bp.payouts.batches.create({
  payouts: [{ amount: 500000, destination: { type: 'alias', alias: '0981600123' },
external_id: 'nom-oct-001' }],
}, 'idem-key-unica-por-lote');
await bp.payouts.batches.get(lote.id);

// Webhooks: el handler verifica firma + frescura, descarta duplicados y solo
// invoca onPaid con eventos válidos. Adapters express() y fastify().
const webhook = bp.webhooks.handler({ secret, onPaid: async ({ reference }) => { /*
entregar */ } });
app.post('/webhooks/bepago', express.raw({ type: '*/*' }), webhook.express());
```

3.2 PHP (7.4+, sin dependencias)

Instalá con `composer require bepago/sdk` (PSR-4, namespace `Bepago\BePago`). Código fuente y issues: github.com/bepago/sdk-php. ¿Sin Composer? Bajá `BePago.php` (el mismo archivo del paquete) y usá `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');  // estado
verificado

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

// Lote (hasta 1000 pagos): queda pendiente de aprobación en el panel salvo auto-
aprobación
$lote = $bp->payouts()->batches()->create([
    'payouts' => [['amount' => 500000, 'destination' => ['type' => 'alias', 'alias' =>
'0981600123'], 'external_id' => 'nom-oct-001']],
], 'idem-key-unica-por-lote');
$bp->payouts()->batches()->get($lote['id']);

// Webhooks: el handler verifica firma + frescura, descarta duplicados y solo
// invoca onPaid con eventos válidos. process() devuelve status/body para PSR-7.
$wh = $bp->webhooks()->handler(['secret' => $secret, 'onPaid' => fn($c) => /* entregar */
null]);

// Verificación puntual de UNA entrega (si no usás el handler):
$evt = BePago::verifyWebhook($secret, ['id' => $id, 'timestamp' => $ts, 'signature' =>
$sig, 'body' => $body]);
```

3.3 Otros lenguajes

La firma HMAC v2 se implementa en ~15 líneas en cualquier lenguaje. La sección Autenticación tiene snippets copy-paste en Node, PHP y Python con un vector de prueba para validar tu implementación.

3.4 Ejemplo oficial

La **tienda demo** es una integración end-to-end mínima (solo `node:http` + el SDK): crea un cobro, embebe el checkout por iframe, vuelve por `return_url` y recibe los webhooks firmados. Descargá el servidor completo: `tienda-demo.mjs`.

4 Autenticación

Dónde se configura. Panel › Integraciones › Claves de API.

Cada request a la API se firma con HMAC-SHA256 (versión 2). No hay tokens Bearer ni OAuth: cada request lleva su firma, calculada con tu secret key (`sk_`).

4.1 Headers requeridos en cada request

HEADER	VALOR	DESCRIPCIÓN
`X-Api-Key`	`pk_test_…`	Tu Key ID (identifica al comercio)
`X-Timestamp`	`1698000000000`	Epoch en milisegundos. Ventana: ±5 min.
`X-Nonce`	`a1b2c3d4e5f6…`	Aleatorio, 16-128 chars `[A-Za-z0-9_-]`. Uno nuevo por request.
`X-Signature-Version`	`2`	Siempre 2.
`X-Signature`	`cf24f5e5…`	HMAC-SHA256 hex de la cadena canónica.
`X-Idempotency-Key`	`pago-001`	Solo en `POST /v1/payouts` (mín. 8 chars).

4.2 Cadena canónica

formato (cada campo en una línea)

```
v2
{timestamp}
{nonce}
{METHOD}
{path_con_query}
{idempotency_key_o_vacio}
{sha256_hex_del_body}
```

Si el request no tiene body (GET), el SHA-256 es el del string vacío: `e3b0c44…` .

Nota. El `path` que se firma es el de la API (`/v1/...` , con su query string si lo hay), **sin** el prefijo `/api` que lleva el dominio. Firmás `/v1/charges` aunque el request vaya a `https://bepago.com/api/v1/charges` .

4.3 Implementación — 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 Implementación — 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 Implementación — 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}
```

Importante. Al mandar el request tenés que enviar **los mismos bytes que hasheaste**. Con `requests` usá `data=body_str` (el string que firmaste), **no** `json=body` : `json=` vuelve a serializar el objeto (espacios, orden) y la firma deja de coincidir.

4.6 Vector de prueba

Usá estos valores fijos para validar tu implementación. Si tu función devuelve la misma firma, funciona:

Entrada

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

Firma esperada

```
SHA-256 del body:  ac687e6ba5c15ff065b1850b4d5860d49bfffe79a5fa3e368bb34044e1e52f42
X-Signature:       cf24f5e57bf986bf34801f49b9b1ee8b3ccf0dc39ced32fb8c10ba1c0082ea65
```

5 Cobros

Un cobro representa un checkout para que tu cliente pague. Creás el cobro por API, recibís una URL, la embebés en un iframe, y el pagador elige su medio de pago.

5.1 Crear un cobro

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

CAMPO	TIPO	REQUERIDO	DESCRIPCIÓN
`reference`	string	Sí	Tu referencia única (pedido, orden). Misma referencia = mismo cobro (idempotente).
`currency`	string	Sí	Siempre `PYG`. Otro valor devuelve `400 validation_error`.
`amount`	integer	Sí	Monto en guaraníes, entero positivo. El mínimo lo fija el medio de pago (QR SIP: Gs 1.000).
`description`	string	Sí	Descripción visible en el checkout (max 140 chars)
`expires_in`	integer	No	Segundos de vida del checkout (60-604800, default 1800 = 30 min)
`return_url`	string	No	URL de retorno post-pago. Debe estar en tus dominios registrados.
`customer`	object	No	`name`, `email`, `document`, `phone`
`metadata`	object	No	Datos arbitrarios. Se guardan con el cobro; no viajan en el webhook ni en la respuesta.

Nota. La `reference` es idempotente: si mandas la misma referencia dos veces, recibís el mismo cobro (200 en vez de 201). Así no se crean cobros duplicados por retries.

5.2 Consultar un cobro

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

Podés usar tu `reference` o el `id` de bePago.

5.3 Listar cobros

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

Devuelve tus cobros del más nuevo al más viejo, paginados por cursor. Parámetros de query opcionales: `limit` (1 a 100, default 20), `status` (filtra por estado), `from` y `to` (rango sobre `created_at`, ISO 8601 o `YYYY-MM-DD`) y `cursor` (el `next_cursor` de la página anterior).

Respuesta

```
{
  "object": "list",
  "livemode": false,
  "data": [ /* cobros, mismo formato que GET /v1/charges/:ref */ ],
  "has_more": true,
  "next_cursor": "chg_..."
}
```

Para traer la página siguiente, repetís el request con `cursor=<next_cursor>`. Cuando `has_more` es `false`, no quedan más resultados.

5.4 Reembolsar

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

El campo `reason` es **obligatorio** (3 a 200 caracteres): queda registrado en la auditoría del reembolso.

Body

```
{ "reason": "Cliente pidió la devolución" }
```

5.5 Cancelar

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

Cancela un cobro que todavía **no se pagó** (`created` o `pending`): el checkout se vence y no se acredita nada. Un cobro ya pagado no se cancela, se reembolsa (devuelve `409`). No lleva body.

5.6 Saldo cobrado

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

Devuelve el saldo disponible y las comisiones pendientes de cobro, en guaraníes.

Respuesta

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

5.7 Estados de un cobro

ESTADO	SIGNIFICADO	ES FINAL
`created`	Cobro creado, checkout activo. El pagador todavía no hizo nada.	No
`pending`	El pagador inició el pago (QR generado, esperando confirmación).	No
`paid`	Pago confirmado. Podés entregar el producto.	No (pasa a settled, o a refunded si lo reembolsás)
`settled`	El dinero está disponible para retirar.	No (todavía puede pasar a refunded)
`expired`	El checkout venció sin pago.	Sí
`refunded`	Reembolsado.	Sí
`failed`	Falló el cobro.	Sí

6 Checkout embebido

Dónde se configura. Panel › Integraciones › Dominio del checkout.

El checkout de bePago es un iframe que cargás en tu web. El pagador elige el medio de pago; vos no tocás datos sensibles.

Checklist. Antes de embeber el checkout, repasá el checklist de integración segura del Quickstart: la orden se completa solo por webhook verificado o por API, el monto sale de tu servidor y el webhook se verifica y deduplica.

6.1 Embeber con bepago.js (recomendado)

El script `bepago.js` renderiza el iframe y te avisa cuando el comprador terminó la pantalla. Pedí el `checkout_url` a tu backend (que es quien crea el cobro con el monto del servidor) y embebelo.

HTML + bepago.js

```
<div id="bepago-checkout"></div>
<script src="https://bepago.com/sdk/bepago.js"></script>
<script>
  BePago.checkout({
    url: checkout_url,          // la checkout_url que devolvió TU backend
    mount: '#bepago-checkout',
    onComplete: ({ reference }) => {
      // solo UI y redirección; la confirmación es del backend
      location.href = '/gracias/' + reference;
    }
  });
</script>
```

Seguridad. `onComplete` significa “el comprador cerró el checkout”, **no** “el pago está confirmado”. Por eso solo recibís `reference` y ningún estado: quién decide si entregás es tu servidor, por webhook verificado con el SDK o con `bp.charges.confirm(reference)`. Nunca entregues desde el navegador.

6.2 Dominios permitidos

Por seguridad, el iframe solo se puede cargar desde los dominios que registraste en **Integraciones › Dominio del checkout** en tu panel. El header `frame-ancestors` restringe a esos orígenes exactos.

6.3 Medios de pago

El cobro se crea en guaraníes. Los medios disponibles para un comercio los habilita bePago; el checkout muestra solo los habilitados.

MEDIO	RED	MONEDA DEL PAGADOR	VOS RECIBÍS EN
QR SIP	SIP · BCP	Guaraníes (PYG)	PYG
QR AR	QR interoperable	Pesos argentinos (ARS)	PYG
Pix	Pix (BCB)	Reales (BRL)	PYG

El pagador paga en la moneda de su medio; vos recibís siempre guaraníes, por el monto del cobro.

7 Pagos Masivos

Dispersá pagos a cualquier cuenta bancaria o billetera de Paraguay por clearing bancario.

7.1 Crear un pago

`POST /v1/payouts` — scope: `payouts:write`

> **Importante.** Cada pago requiere un header `X-Idempotency-Key` único (min 8 chars). Si retransmitís el mismo request con la misma key, recibís el mismo pago (`200` en vez de `201`) sin crear uno nuevo.

7.2 Destino por cuenta bancaria

Body

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

7.3 Destino por alias

Body

```
{
  "amount": 500000,
  "currency": "PYG",
  "destination": {
    "type": "alias",
    "alias": "0981600123"
  },
  "description": "Pago a proveedor"
}
```

El alias puede ser un número de teléfono, CI o email. bePago lo resuelve contra el directorio SIPAP y confirma el nombre del titular.

7.4 Consultar

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

7.5 Listar pagos

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

Devuelve tus pagos del más nuevo al más viejo, paginados por cursor. Mismos parámetros de query que el listado de cobros: `limit` (1 a 100, default 20), `status`, `from`, `to` y `cursor`. La respuesta trae `data`, `has_more` y `next_cursor` con la misma mecánica de paginación.

7.6 Cancelar

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

Solo se puede cancelar si el pago no salió al banco (`ready` o `held`).

7.7 Saldo

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

Respuesta

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

7.8 Estados

ESTADO	SIGNIFICADO	ES FINAL
`created`	Recién creado	No
`validating`	Validando datos del destinatario	No
`ready`	En cola, sale al banco por clearing	No
`held`	Retenido: un supervisor tiene que aprobarlo	No
`submitting`	Enviando al banco	No
`submitted`	El banco está procesando	No
`unknown`	Sin confirmación del banco. Plataforma lo verifica. **NO reintentar.**	No
`completed`	Acreditado en la cuenta del destinatario	Sí
`failed`	Falló: la plata volvió al saldo	Sí
`rejected`	Rechazado por un supervisor	Sí
`cancelled`	Cancelado antes de salir al banco	Sí
`expired`	Nadie lo aprobó a tiempo	Sí

Seguridad. Si un pago queda en `unknown`, **NO lo reintentar**. Plataforma lo verifica manualmente. Si lo reintentás podés pagar dos veces.

7.9 Lotes (dispersión masiva)

Para pagar muchos destinos de una sola vez hay dos caminos: la **API** (tu sistema carga y ejecuta el lote en una llamada) o el **panel** (subís un CSV, lo revisás y otro usuario lo aprueba — cuatro ojos).

Por API

`POST /v1/payouts/batches` — scope: `payouts:write`. Con el SDK: `bp.payouts.batches.create(body, idempotencyKey)` (Node) o `$bp->payouts()->batches()->create($body, $idempotencyKey)` (PHP); el estado con `batches.get(id)`.

Importante. El lote entero lleva un header `X-Idempotency-Key` único (mín. 8 chars). Además, cada pago del lote es idempotente por su posición (`batch:<id>:<fila>`): si retransmitís el mismo lote, **jamás se paga dos veces**. El saldo se valida para el lote **completo** antes de crear ningún pago.

Nota. Cuatro ojos por API: por defecto, un lote cargado por API queda `pending_approval` y necesita que un usuario del panel lo apruebe (distinto de quien tiene la clave). Para habilitar la auto-aprobación, pedí a soporte que active la opción en tu comercio; además, la clave necesita el scope `payouts:approve` y tener **al menos una IP en su lista de IPs permitidas**. Sin las tres condiciones, el lote siempre espera aprobación manual.

body

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

Respuesta


```
{
  "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": "nom-oct-001", "operation_number": null, "failure_code": null,
"beneficiary_name": null },
    { "row": 1, "id": "pay_...", "status": "ready", "amount": 3200000, "fee": 38400,
"net": 3238400, "reference": "nom-oct-002", "operation_number": null, "failure_code":
null, "beneficiary_name": "Maria Cabrera" }
  ]
}
```


El lote nace `pending_approval` y pasa a `processing` cuando se aprueba (manual o automática). Cuando cada pago llega a un estado final queda `completed`, o `completed_with_errors` si alguna fila falló; un lote rechazado antes de ejecutarse queda `cancelled`. Cada pago individual sigue los mismos estados de un pago suelto.

`GET /v1/payouts/batches/:id` — scope: `payouts:read` — devuelve el mismo cuerpo, con el estado actualizado de cada fila (número de operación y titular confirmado cuando se acreditan). Los importes de comisión del ejemplo son ilustrativos: `fee` y `net` salen de la tasa de tu contrato.

> **Nota.** Si un pago del lote falla, los demás siguen su curso y el monto del fallido vuelve al saldo con su asiento contable. Un lote nunca se paga a medias por saldo: o alcanza para todo, o no se crea nada.

Por panel (CSV)

En **Lotes (CSV)** del panel subís un archivo, ves la vista previa con las filas válidas e inválidas (y el motivo de cada una), el monto total y lo que descuenta del saldo. Nada se paga hasta que **otro usuario** (quien cargó no puede aprobar su propio lote) lo aprueba. Al terminar podés exportar el resultado a CSV con el estado, número de operación y titular confirmado de cada pago.

El CSV tiene una fila de encabezado con estas columnas (hay una plantilla descargable en la misma pantalla):

COLUMNA	PARA QUÉ	REQUERIDA
`tipo`	`alias` o `cuenta`	Sí
`alias`	Teléfono, CI o email (si `tipo=alias`)	Si es alias
`banco`	Código de banco SIPAP (si `tipo=cuenta`)	Si es cuenta
`cuenta`	Número de cuenta	Si es cuenta
`documento`	CI o RUC del titular	Si es cuenta
`titular`	Nombre del titular	Si es cuenta
`monto`	Monto en guaraníes (entero)	Sí
`referencia`	Tu identificador de la fila	No
`descripcion`	Texto libre	No

ejemplo CSV

```
tipo,alias,banco,cuenta,documento,titular,monto,referencia,descripcion
alias,0981600123,,,,,500000,nom-oct-001,Quincena octubre
cuenta,,0017,0012345678,4567890,Maria Cabrera,3200000,nom-oct-002,Quincena octubre
```

Nota. En **modo prueba** todo el flujo del lote funciona igual contra el sandbox: los pagos no tocan ningún banco. Un alias o cuenta que contenga `invalido` (o una cuenta que empiece en `9999`) hace fallar esa fila a propósito, para probar el camino de fallas parciales. Ver Sandbox.

8 Comisiones

La comisión de tu comercio es la acordada en tu contrato con bePago y la ves aplicada en tu panel: en cada cobro y cada pago, en el saldo del día y en los extractos. No la calcules en tu sistema: cada cobro y cada pago devuelven los campos fee y net ya calculados, y el webhook los incluye. Esta sección explica cómo se aplican y cómo se redondean para que tu conciliación cuadre al guaraní.

En un cobro, `fee` es la comisión y `net` lo que se acredita a tu saldo (`amount - fee`); `expected_net` es el neto previsto al crearlo: la tasa de cobro depende del medio (QR SIP, Pix, QR Argentina), así que hasta que el pagador lo elige se estima con el de mayor comisión, y al elegirlo queda fija. En un pago, `fee` es la comisión y `net` lo que sale de tu saldo (`amount + fee`). Los lotes traen `fee` y `net` por fila y `total_fee` del lote. Los webhooks `charge.*` y `payout.*` llevan los mismos campos que la API.

La comisión se cobra en uno de dos momentos, según cómo esté configurado tu comercio:

MODO	CUÁNDO SE COBRA	CÓMO SE CALCULA
`per_transaction`	En cada pago	Un fijo opcional más un porcentaje del monto del pago
`on_funding`	Al acreditar un fondeo	Un porcentaje del depósito (sin fijo). Los pagos salen sin comisión.

El porcentaje se expresa en **puntos básicos** (bps): 1 bps es una centésima de punto porcentual, 10.000 bps es el 100 %. La comisión se devenga con su propio asiento contable en el mismo momento del movimiento, así que lo que ves en el pago es exactamente lo que se registra.

8.1 Regla de redondeo

Los guaraníes no tienen decimales. Cuando el porcentaje da una fracción, la comisión se redondea **siempre hacia arriba** al guaraní entero. Es una sola regla, idéntica para pagos y cobros:

Redondeo de la comisión (hacia arriba al guaraní), con T = tasa de tu contrato en puntos básicos

```
fee = ceil(monto × T / 10000) + fijo
# en cobros, si tu contrato tiene mínimo: fee = max(fee, mínimo)
```

Cuando el cálculo da una fracción, sube al entero siguiente; cuando da un entero exacto, no sube de más. Una fila de muestra:

MONTO	TASA	MONTO × T / 10000	FEE
Gs 155.555	T = 137 bps (tasa de ejemplo, no es tu tarifa)	2.131,1035	Gs 2.132

Nota. El redondeo siempre favorece al monto entero superior: así la comisión nunca queda en fracciones de guaraní y el asiento contable cuadra al guaraní.

9 Webhooks

Dónde se configura. Panel › Integraciones › Webhook.

bePago envía webhooks firmados a tu endpoint cuando un recurso cambia de estado. Los configurás desde **Integraciones › Webhook** en tu panel.

9.1 Eventos

EVENTO	MÓDULO	CUANDO SE DISPARA
`charge.paid`	Cobros	El cobro fue pagado
`charge.settled`	Cobros	El dinero está disponible para retirar
`charge.expired`	Cobros	El checkout venció sin pago
`charge.refunded`	Cobros	Reembolsado
`charge.failed`	Cobros	Falló
`payout.processing`	Pagos	El pago se está procesando
`payout.completed`	Pagos	Acreditado
`payout.failed`	Pagos	Falló
`payout.held`	Pagos	Retenido (necesita aprobación)
`payout.cancelled`	Pagos	Cancelado
`payout.expired`	Pagos	Retenido sin decisión a tiempo: venció y el monto volvió al saldo
`payout.unknown`	Pagos	Sin confirmación del banco

9.2 Payload

Ejemplo charge.paid

```
{
  "id": "clx8y2z0001...",
  "event": "charge.paid",
  "livemode": true,
  "merchant_id": "1042",
  "data": {
    "id": "chg_abc123",
    "reference": "pedido-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` es tu ID de comercio, un número (el mismo que ves en el panel como "ID de comercio: 1042"). Los importes del ejemplo son ilustrativos: `fee` y `net` salen de la tasa de tu contrato. El webhook trae el mismo `net` del ledger que la API y el panel: en `charge.paid` coincide con `expected_net`; en `charge.refunded` baja (el reembolso sale del saldo) y en `charge.expired` es `0`.

9.3 Verificar la firma

Cada webhook lleva `X-Webhook-Timestamp`, `X-Webhook-Signature` y `X-BePago-Event-Id`. La firma es `HMAC-SHA256(webhook_secret, timestamp + body)`. Verificá **la firma y la frescura del timestamp** (rechazá entregas fuera de ±5 minutos): así un webhook viejo capturado y reenviado no pasa.

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; // frescura
±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; //
frescura ±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:  # frescura ±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 y deduplicación

Cada evento trae un `id` único (en el cuerpo y en el header `X-BePago-Event-Id`). Los reintentos reenvían el **mismo id** con un timestamp fresco, así que la firma por sí sola no distingue un reintento legítimo de un duplicado. Guardá los ids ya procesados y descartá los repetidos para que tu procesamiento sea idempotente.

Node

```
const eventId = req.headers['x-bepago-event-id'];
if (await yaProcesado(eventId)) return res.sendStatus(200); // duplicado
await marcarProcesado(eventId);
// ... procesá el evento una sola vez
```

9.5 Reintentos

Si tu endpoint no responde 2xx, bePago reintenta con backoff creciente (1min, 5min, 15min, 1h, 6h, 12h, 24h, 48h, 72h): 9 intentos a lo largo de hasta 3 días, para tolerar caídas prolongadas del servidor de tu comercio. Después del último intento fallido, la entrega se marca como fallida y queda en el panel para reenvío manual. El `id` del evento es estable entre reintentos.

Nota. Un webhook perdido no rompe nada. Siempre podés consultar el estado por API. Los webhooks son una comodidad, no la fuente de verdad.

10 Errores

Todas las respuestas de error siguen la misma estructura:

formato de error

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

El campo `action` te dice que hacer: `retry`, `fix_input`, `reauthenticate`, `contact_admin`, `contact_platform`, `wait` o `none`.

10.1 Códigos comunes

CÓDIGO	HTTP	QUÉ HACER
`unauthorized`	401	Falla de autenticación. `details.reason` dice cuál: `bad_signature` (revisá la cadena canónica), `expired` (timestamp fuera de ±5 min, sincronizá el reloj), `replayed_nonce` (generá un nonce nuevo), `ip_not_allowed`, `key_revoked`, `unknown_key`.
`signature_version_unsupported`	401	Mandá `X-Signature-Version: 2`
`insufficient_scope`	403	La key no tiene el scope necesario (`details.required_scope`). Creá una key que lo incluya.
`module_not_enabled`	403	Tu comercio no tiene habilitado ese módulo
`validation_error`	400	Revisá los campos del body
`insufficient_funds`	422	No hay saldo. Contactá a plataforma para fondear.
`limit_exceeded`	422	El pago supera un límite configurado en tu comercio
`missing_idempotency_key`	400	Agregá X-Idempotency-Key en POST de pagos y lotes (mín. 8 chars)
`idempotency_conflict`	409	Esa idempotency key ya se usó con otro contenido
`rate_limited`	429	Esperá y reintentá
`not_found`	404	El recurso no existe
`not_cancellable`	409	El pago ya salió al banco o está cerrado
`charge_not_refundable`	409	El cobro no se puede reembolsar en su estado actual
`checkout_closed`	409	El cobro ya no admite pago ni cancelación (pagado, vencido o cancelado)
`checkout_domain_invalid`	422	Usá el origen exacto con https al registrar el dominio del checkout
`duplicate_suspect`	409	Posible pago duplicado (mismo destino y monto reciente)

11 Seguridad

11.1 Protección de claves

- La `sk_` (secret key) NUNCA va en el cliente (browser, app móvil, iframe). Solo en tu server.
- La `pk_` (Key ID) identifica tu comercio pero no puede firmar. Se puede exponer.
- Si un secreto se filtró o lo perdiste: creá una key nueva, cambiala en tu servidor y revocá la anterior desde el panel. Una key revocada deja de autenticar al instante.
- Lista de IPs permitidas: opcional, pero recomendada en producción. Con la lista cargada, un request desde otra IP se rechaza aunque la firma sea válida.

11.2 Anti-replay

- Ventana de timestamp: ±5 minutos. Sincronizá el reloj de tu server con NTP.
- El nonce se consume en el primer uso. Una firma capturada no se puede reenviar.
- La idempotency key va firmada: no se puede cambiar para duplicar un pago.

11.3 Aislamiento test/live

- Las claves de prueba y producción viven en universos completamente separados.
- Un pago de prueba NUNCA puede contarse como real.
- El campo `livemode` en cada respuesta indica si el recurso es real o de prueba.
- El modo sale siempre de la clave, nunca del body.

11.4 Checkout (iframe)

- `frame-ancestors` restringido a tus dominios registrados.
- Nunca pases datos sensibles por query string en la checkout_url.
- Validá el `origin` de los postMessage antes de procesarlos.

11.5 Webhooks

- Siempre verificá la firma HMAC antes de procesar un webhook.
- Siempre confirmá el estado por API después de recibir un webhook. El webhook es una notificación, no la fuente de verdad.
- Usá `timingSafeEqual` (Node) o `hash_equals` (PHP) para comparar firmas (evita timing attacks).

12 Sandbox (datos de prueba)

Dónde se configura. Panel › Integraciones › Cobro de prueba.

Con tus claves `sk_test_` podés probar toda la integración sin mover plata real.

12.1 Cobros

Primero necesitás un cobro de prueba. Lo creás por API (`POST /v1/charges` con tu clave `sk_test_`) o, sin escribir código, desde el panel en **Integraciones › Cobro de prueba**.

Con clave test tenés dos formas de simular el pago de un cobro, las dos equivalentes:

1. **Desde el checkout de prueba:** abrí la URL del cobro y elegí un medio de pago. El checkout de prueba no cobra nada real — al elegir el medio, el cobro se marca como pagado igual que en producción.
2. **Por API:** con el endpoint de sandbox, sin abrir el checkout.

Simular pago por API (solo sk_test)

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

Cualquiera de las dos transiciona el cobro a `paid` , crea los asientos de ledger (el saldo sube) y dispara el webhook `charge.paid` . Así podés probar el ciclo completo sin mover plata real.

Ejemplo

```
const charge = await bp.charges.create({ reference: 'test-1', currency: 'PYG', amount:
50000, description: 'Test' });
// Simular el pago
await fetch(BASE + '/v1/test/charges/test-1/pay', { method: 'POST', headers: {
...hmacHeaders } });
// Ahora el cobro está en 'paid' y el saldo aumentó
```

Un cobro cuyo checkout nunca se abre queda en `created` : crear el cobro solo genera la URL de pago, no mueve plata. Mientras nadie abra el checkout ni llames al endpoint de sandbox, el cobro se queda en `created` (hasta vencer). Al elegir el medio de pago pasa a `pending` (QR emitido) y, al confirmarse, a `paid` .

12.2 Pagos masivos

Los pagos de prueba simulan el ciclo completo (validación, envío, resultado) sin tocar ningún banco. Un destino con `invalido` en el alias, o una cuenta que empieza en `9999` , termina en `failed` a propósito; el resto se acredita.

12.3 Webhooks

Se envían igual en test y live. El campo `livemode: false` los distingue. Tu endpoint los recibe de la misma forma.

12.4 Pasar a producción

1. Cambiá tus claves de `sk_test_` a `sk_live_`.
2. Cargá la lista de IPs permitidas en tu key live (opcional, recomendado).
3. Registrá tus dominios del checkout.
4. El mismo código funciona: no hay nada que reescribir.

13 Referencia de la API

13.1 Cobros

MÉTODO	ENDPOINT	SCOPE	DESCRIPCIÓN
POST	/v1/charges	collect:write	Crear un cobro (idempotente por reference)
GET	/v1/charges/:ref	collect:read	Consultar (por reference o ID)
GET	/v1/charges	collect:read	Listar (cursor + filtros status/fecha)
POST	/v1/charges/:ref/refund	collect:write	Reembolsar
POST	/v1/charges/:ref/cancel	collect:write	Cancelar (solo created/pending)
GET	/v1/charges/balance	collect:read	Saldo cobrado y comisiones pendientes
POST	/v1/test/charges/:ref/pay	collect:write	Simular el pago (solo sk_test)

13.2 Pagos Masivos

MÉTODO	ENDPOINT	SCOPE	DESCRIPCIÓN
POST	/v1/payouts	payouts:write	Crear pago (requiere X-Idempotency-Key)
GET	/v1/payouts/:id	payouts:read	Consultar pago
GET	/v1/payouts	payouts:read	Listar (cursor + filtros status/fecha)
POST	/v1/payouts/:id/cancel	payouts:write	Cancelar (solo ready/held)
POST	/v1/payouts/batches	payouts:write	Lote de pagos (requiere X-Idempotency-Key)
GET	/v1/payouts/batches/:id	payouts:read	Consultar lote
GET	/v1/balance	balance:read	Saldo disponible y reservado

14 Límites y rate limiting

LÍMITE	VALOR
Requests por minuto (por key)	600
Requests por minuto (por IP)	300
Ventana de timestamp	±5 minutos
Nonce	16-128 chars, un uso
Monto máximo por request	Gs 1.000.000.000.000
Pagos por lote	1.000
Idempotency key	Mín. 8 chars
Body máximo	512 KB

Si superás el rate limit recibís `429 rate_limited`. Esperá y reintentá con backoff exponencial.

¿Necesitás ayuda con la integración? Escribinos a hola@bepago.com