EcomLabs

STEP-BY-STEP GUIDE

# Idempotency keys for payments in Node

The full Node example, client and server, so a retry, a double click or a timeout charges exactly once. With a curl demo that shows every case, and what is left for production.

| | |
|---|---|
| FOR | Checkouts and payment APIs in Node.js |
| TESTED WITH | Node 22.22.3 · curl 8.19 · Windows 11 |
| UPDATED | October 30, 2026 |
| DOCS | docs.stripe.com/api/idempotent_requests |
| ARTICLE | ecomlabs.dev/g/pagos-en |
| AUTHOR | Luis Morales · @luismoralesdev |

# Contents


| | |
|---|---|
| Before you start | What you will build |
| Step 01 | The server |
| Step 02 | The client |
| Step 03 | Try the demo with curl |
| Step 04 | Before you take it to production |
| Final | Checklist |

BEFORE YOU START

## What you will build

A Node payment server that stores every `Idempotency-Key` with a fingerprint of the body and the response, and a client that creates one key per payment attempt and repeats it on every retry. With both, a double click, a retry or a timeout charges once. Everything uses Node built-ins: nothing to install.

We tested it on October 30, 2026 with Node 22.22.3 and curl 8.19 on Windows 11. The client needs global `fetch`, `AbortSignal.timeout` and `crypto.randomUUID()`: every current browser has them (in a secure context, HTTPS), and so does Node 22.

STEP 01

## The server

Save this as `server.mjs`. It takes the function that charges (`charge`), so you can plug in your real provider or a fake one for testing:

```
import { createServer } from "node:http";
import { createHash } from "node:crypto";

export function createPaymentServer(charge) {
  const keys = new Map();

  const send = (res, status, body, headers = {}) => {
    res.writeHead(status, { "content-type": "application/json", ...headers });
    res.end(JSON.stringify(body));
  };

  return createServer(async (req, res) => {
    if (req.method !== "POST" || req.url !== "/payments") return send(res, 404, { error: "not found" });

    const key = req.headers["idempotency-key"];
    if (!key || key.length > 255) return send(res, 400, { error: "Idempotency-Key is missing or too long" });

    let raw = "";
    for await (const chunk of req) raw += chunk;
    let payment;
    try {
      payment = JSON.parse(raw);
    } catch {
      return send(res, 400, { error: "body is not JSON" });
    }
    if (!Number.isInteger(payment?.amount) || payment.amount <= 0 || !/^[a-z]{3}$/.test(payment.currency)) {
      return send(res, 400, { error: "amount (integer, cents) and currency (e.g. mxn) are required" });
    }
    const fingerprint = createHash("sha256").update(raw).digest("hex");
```

```
    const saved = keys.get(key);
    if (saved) {
      if (saved.fingerprint !== fingerprint) return send(res, 422, { error: "Idempotency-Key already used with another body" });
      if (!saved.done) return send(res, 409, { error: "a request with this Idempotency-Key is in progress" });
      return send(res, saved.status, saved.body, { "idempotent-replayed": "true" });
    }

    keys.set(key, { fingerprint, done: false });

    let status, body;
    try {
      body = await charge(payment);
      status = 201;
    } catch {
      status = 500;
      body = { error: "payment outcome unknown; check the order before trying again" };
    }
    keys.set(key, { fingerprint, done: true, status, body });
    send(res, status, body);
  });
}
```

Order matters: validate, fingerprint, look the key up and claim it before charging. There is no `await` between `keys.get` and `keys.set`, so two requests with the same key cannot both get through.

---

STEP 02

## The client

Save this as `client.mjs`. `postPayment` retries with the same key; `onPay` is what you wire to the button:

```
const sleep = (ms) => new Promise((resolve) => setTimeout(resolve, ms));

export async function postPayment(url, payment, key, { retries = 3, timeoutMs = 10000 } = {}) {
  for (let attempt = 0; ; attempt++) {
    try {
      const res = await fetch(url, {
        method: "POST",
        headers: { "content-type": "application/json", "idempotency-key": key },
        body: JSON.stringify(payment),
        signal: AbortSignal.timeout(timeoutMs),
      });
      if ([409, 502, 503, 504].includes(res.status) && attempt < retries) {
        await sleep(200 * 2 ** attempt);
        continue;
      }
      return { status: res.status, replayed: res.headers.get("idempotent-replayed") === "true", body: await res.json() };
    } catch (err) {
```

```
      if (attempt >= retries) throw err;
      await sleep(200 * 2 ** attempt);
    }
  }
}

let key = null;

export async function onPay(payment, url = "/payments") {
  key ??= crypto.randomUUID();
  const result = await postPayment(url, payment, key);
  if (result.status < 500 && result.status !== 409) key = null;
  return result;
}
```

The rule: the key is created on the attempt's first click and dropped only after a final answer (a `201`, or a `4xx` other than `409`). After a network error, a `409` or a `5xx` it is kept.

---

STEP 03

## Try the demo with curl

Save `demo.mjs` and start it with `node demo.mjs`:

```
import { createPaymentServer } from "./server.mjs";

let charges = 0;
const charge = async ({ amount, currency }) => {
  await new Promise((resolve) => setTimeout(resolve, 300));
  return { id: `ch_${++charges}`, amount, currency, status: "succeeded" };
};

createPaymentServer(charge).listen(8787, () => console.log("POST http://localhost:8787/payments"));
```

In another bash terminal (Git Bash on Windows) save and run `bash curl.sh`:

```
KEY=8e03978e-40d5-43e8-bc93-6894a57f9324
BODY='{"orderId":"A-1001","amount":129900,"currency":"mxn"}'
pay() { curl -s http://localhost:8787/payments -H "content-type: application/json" -H "Idempotency-Key: $1" -d
"$2" -w '  -> %{http_code} %header{idempotent-replayed}\n'; }

pay $KEY "$BODY"
pay $KEY "$BODY"
pay $KEY '{"orderId":"A-1001","amount":1,"currency":"mxn"}'
pay other-key "$BODY" & pay other-key "$BODY"; wait
```

This is the output we got. The last two lines can come out in either order:

```
{"id":"ch_1","amount":129900,"currency":"mxn","status":"succeeded"}  -> 201
{"id":"ch_1","amount":129900,"currency":"mxn","status":"succeeded"}  -> 201 true
{"error":"Idempotency-Key already used with another body"}  -> 422
{"error":"a request with this Idempotency-Key is in progress"}  -> 409
{"id":"ch_2","amount":129900,"currency":"mxn","status":"succeeded"}  -> 201
```

STEP 04

## Before you take it to production

- **A shared store.** The `Map` lives in one process. With several instances, use a table with the key as `PRIMARY KEY` and claim it with `INSERT ... ON CONFLICT (key) DO NOTHING RETURNING key`: only the first request gets the row back. That is our recommendation; the SQL is not part of this guide's tests.
- **Expiry.** Stripe keeps its keys for at least 24 hours. Pick your window, publish it and delete older keys.
- **A key towards the provider.** Stripe accepts `Idempotency-Key` on every `POST`. Send one stable key per attempt so your own retry to Stripe cannot charge twice either.
- **Keys per account.** Store each key with the account that sent it, so nobody receives someone else's stored response.

> **CAUTION**
>
> Do not delete the key when the charge fails with an unknown error. If the provider did charge, the retry charges again. Store the `500` and reconcile the order (webhooks, a lookup at the provider) before starting an attempt with a new key.

FINAL

## Checklist

- [ ] The client creates the key when the payment attempt starts, with `crypto.randomUUID()`.
- [ ] Retries and repeated clicks send the same key and the same body.
- [ ] The server claims the key before charging.
- [ ] A key with another body gets `422`; one still running gets `409`.
- [ ] Replays return the stored response, `500` errors included.
- [ ] Keys live in a shared store and expire under a written policy.
- [ ] Your server also sends a key to the payment provider.
- [ ] `curl.sh` gives `201`, the same response replayed, `422` and a `409` for the simultaneous request.