DEVELOPER TUTORIAL

Building a Replay-Safe Payment Webhook

A walkthrough of FamNest's Flutterwave webhook handler — and the three mistakes it avoids.

System	FamNest — AI wellness coach for parents (Next.js 16 · Supabase · Groq Llama 3.3 70B)
Audience	Backend / full-stack developers integrating any payment webhook
Author	Virginia Mwega — Technical Writer & Documentation Engineer
Source	Annotated from `app/api/webhooks/flutterwave/route.ts`, production code

The problem

A payment webhook is the one endpoint in most apps where a bug doesn't just throw an error — it either gives someone Premium for free, or charges them and gives them nothing. Three things can go wrong, and a real handler has to defend against all three at once:

1. **Spoofing** — anyone can POST a fake "payment successful" body to a public URL if you don't verify it came from your processor.
2. **Trusting the body** — even a correctly-signed webhook is still just a claim. The processor's own API is the source of truth, not the JSON it sent you.
3. **Duplicate or unknown events** — processors retry on timeout, and send event types you didn't plan for. A handler that crashes or double-grants on either is broken in production, just rarely.

This tutorial walks through FamNest's actual handler for Flutterwave (the payment processor used because Stripe doesn't support Kenya), step by step.

Step 1 — Verify the signature, constant-time

Flutterwave signs webhooks differently from Stripe: instead of an HMAC of the body, it sends back the exact secret you configured in the dashboard, in a `verif-hash` header. The handler must still compare it carefully — a naive `===` string comparison leaks timing information that an attacker can use to guess the secret one character at a time.

```
const signature = request.headers.get('verif-hash')
const secret = process.env.FLUTTERWAVE_SECRET_HASH

if (!secret || !signature || !safeEqual(signature, secret)) {
  return new Response('Invalid signature', { status: 401 })
}

function safeEqual(a: string, b: string): boolean {
  const ab = Buffer.from(a)
  const bb = Buffer.from(b)
  if (ab.length !== bb.length) return false
  return crypto.timingSafeEqual(ab, bb)
}
```

> **Why the length check first?** `crypto.timingSafeEqual` throws if the two buffers aren't the same length — it doesn't return `false`. Skipping the length check would crash the handler on a malformed header instead of cleanly rejecting it.

Step 2 — Never trust the webhook body for money

A signed webhook proves the request came from Flutterwave. It does not prove the payment is real, final, or hasn't been altered in transit logic upstream of the signature. The handler treats the webhook purely as a *trigger to go check*, not as the grant decision itself:

```
if (
  event?.event === 'charge.completed' &&
  event.data?.status === 'successful' &&
  event.data.id != null
) {
  const txn = await verifyTransaction(event.data.id)   // hits Flutterwave's API directly
  await activatePremiumFromTransaction(txn)             // grants based on THAT response
}
```

> **The pattern:** webhook says "something happened" → server calls the processor's own API to ask "did it really?" → only the API's answer ever triggers a grant. This is the same re-verification pattern used in the companion `billing/callback` route, so a forged redirect URL is exactly as harmless as a forged webhook body.

Step 3 — Handle duplicates and unknown events gracefully

Both a first purchase and an auto-renewal arrive as the same `charge.completed` event — the handler doesn't need to distinguish them, because `activatePremiumFromTransaction` is idempotent on the transaction id. Events the handler doesn't recognize (like `subscription.cancelled`, handled elsewhere via an in-app cancel action) are acknowledged with `200` and silently ignored:

```
try {
  if (/* charge.completed && successful */) {
    // ...verify and activate
  }
  // Other events are handled in-app via the cancel action — ack and ignore here.
} catch (err) {
  console.error('[flw-webhook] handler error:', err)
  return new Response('Webhook handler error', { status: 500 })
}

return Response.json({ received: true })
```

> **Why return 200 for events you ignore?** Returning a non-2xx for an event type you simply don't act on tells Flutterwave the delivery failed — it will retry, forever, for an event you were never going to handle differently. Acknowledge what you don't use; only fail loudly on events you do.

Full handler, annotated

```
export async function POST(request: Request) {
  // 1. Verify the signature before parsing anything
  const signature = request.headers.get('verif-hash')
  const secret = process.env.FLUTTERWAVE_SECRET_HASH
  if (!secret || !signature || !safeEqual(signature, secret)) {
    return new Response('Invalid signature', { status: 401 })
  }

  // 2. Parse defensively — a bad body shouldn't 500
  let event
  try {
    event = await request.json()
  } catch {
    return new Response('Bad payload', { status: 400 })
  }

  // 3. Re-verify against the processor's API before granting anything
  try {
    if (event?.event === 'charge.completed' &&
        event.data?.status === 'successful' &&
        event.data.id != null) {
      const txn = await verifyTransaction(event.data.id)
      await activatePremiumFromTransaction(txn)
    }
  } catch (err) {
    console.error('[flw-webhook] handler error:', err)
    return new Response('Webhook handler error', { status: 500 })
  }

  // 4. Always ack — including events you don't act on
  return Response.json({ received: true })
}
```

Testing checklist

- ☐ Send a request with no `verif-hash` header → expect `401`.
- ☐ Send a request with a wrong-length `verif-hash` → expect `401`, not a crash.
- ☐ Send malformed JSON with a valid signature → expect `400`.
- ☐ Send a valid `charge.completed` event twice → expect Premium granted once, second call is a no-op.
- ☐ Send a valid signature with an unrecognized event type → expect `200`, no grant, no retry from the processor.

- ☐ Point `verifyTransaction` at a sandbox transaction id that doesn't exist → expect a clean `500`, not an unhandled rejection.

Takeaways

The handler is under 40 lines, but every line is doing exactly one job: reject what isn't signed, re-verify what claims to be money, and acknowledge what you're choosing not to act on. That's the whole pattern — it transfers directly to Stripe, Paddle, Paystack, or any other processor's webhook, because the failure modes (spoofing, blind trust, retry storms) are the same everywhere.

Developer Tutorial — written from the production codebase by Virginia Mwega · virginiamwega2@gmail.com