

PUBLISHER STARTER BACKEND — THE RUNNABLE REDEMPTION KIT

A complete, runnable publisher backend for FlipKey.gg redemption. Express + SQLite + viem. Boots on a laptop, deploys to a VPS or any platform that runs Node, scales to thousands of redemptions before you outgrow the defaults. **Download the kit (.zip).**

Two ways to use this kit:

- 1. **No backend yet?** Unzip, configure `.env`, import your keys, run. You have a working FlipKey.gg integration in 30 minutes.
- 2. **Have your own stack?** Read it as the spec. Every route, every signature check, every DB query is the canonical shape — port it to PHP/Python/Go/Rails/.NET/whatever. See *Reimplementing in your stack* below.

Not sure which, or don't want to run a server at all? Choosing a Key Station compares the hosted station on GameLicense.org, the one-command droplet install (which runs this kit for you), and your own stack, with what each costs.

What you need from FlipKey.gg

When you're approved as a publisher, FlipKey.gg gives you three values:

VALUE	WHAT IT IS	WHERE TO GET IT
<code>FLIPKEY_PUBLISHER_ID</code>	Your numeric publisher ID	Sent to you by FlipKey.gg at onboarding
<code>FLIPKEY_WEBHOOK_SECRET</code>	HMAC secret for signing activation reports + answering link tests	Publisher dashboard → Webhooks
<code>NFT_CONTRACT_ADDRESS</code>	Your FlipKey-deployed NFT contract address	Sent to you by FlipKey.gg at onboarding

You'll also need Node 20+ (Node 22 LTS recommended — run a currently supported LTS), a chain RPC URL (public RPCs work for testing; use a paid endpoint — Alchemy, Infura, QuickNode — in production) and a public HTTPS hostname for the app (e.g., `redeem.yourgame.com`).

5-minute quickstart

```
unzip publisher-starter.zip && cd publisher-starter
cp .env.example .env
# Edit .env — fill in the FlipKey values, your contract address, your hostname

npm install
npm run init-db # creates db/keys.db
npm run import-keys -- --file my-steam-keys.csv \ # bulk-import a CSV
                  --drop 42 \
                  --platform steam
npm start # listens on port 3000
```

That's the whole thing. Reverse-proxy it from your domain, list a game on FlipKey.gg pointing at `https://yourdomain.com/redeem/{tokenId}`, run the link test, and you're live. Prefer containers? `docker compose up --build`.

Configuration (`.env`)

Every variable is documented inline in `.env.example`. Required minimum:

```
FLIPKEY_PUBLISHER_ID=1234
FLIPKEY_WEBHOOK_SECRET=...
FLIPKEY_BASE_URL=https://flipkey.gg           # https://dev.flipkey.gg for sandbox
NFT_CONTRACT_ADDRESS=0x...
CONTRACT_TYPE=ERC1155                        # FlipKey v1 licenses are ERC-1155
CHAIN_RPC_URL=https://mainnet.base.org        # paid endpoint in production
CHAIN_ID=8453                                # 8453 = Base mainnet; 84532 = Base Sepolia (sandbox)
PUBLIC_BASE_URL=https://redeem.yourgame.com
PUBLISHER_NAME=Your Studio
```

Database: SQLite out of the box (`DATABASE_PATH=...` to move the file). Already running MySQL or Postgres? Two lines:

```
DB_DRIVER=mysql                             # or: postgres
DATABASE_URL=mysql://user:pass@localhost:3306/redeem
```

Every script and route works identically on all three engines — same commands, same behavior, dialect-correct SQL underneath (`npm run init-db` applies the right schema automatically). MySQL needs 8.0+, Postgres 9.5+ (both for `SKIP LOCKED` on the atomic claim).

Adding keys

Bulk import from CSV:

```
npm run import-keys -- --file steam-na.csv --drop 42 --platform steam --region NA
```

The CSV must have a column named `key` (case-insensitive); other columns are ignored. `--region` and `--edition` are optional but recommended when you have keys with restrictions — the redemption handler matches on `(drop_id, platform, region, edition)` when claiming a key. Region `WW` means "any region acceptable."

Counting inventory: `node scripts/list-inventory.js` — tokens, key buckets, recent activations, on any DB engine.

Paste-in upload page (no shell needed): set `VAULT_UPLOAD_PASSWORD` (16+ characters) and `/vault/upload` turns on — HTTP Basic auth as user `vault`, a product picker that lists your FlipKey listings (one per platform) and shows each one's region, price, sales and how many keys still cover unsold copies, and a paste box that takes one key per line or a CSV with a `key` column. Keys are sealed before they touch disk (see "Keys at rest" below); the result shows counts and the last four characters of each line, never a whole key; repeats are rejected; ten failed sign-ins from one address lock it for 15 minutes; nothing is logged. Leave the variable unset and the page does not exist. Handy when the person loading keys is not the person running the box. You still run the box: this kit is yours to host. If you'd rather not host anything, our partner GameLicense.org offers paid hosting (\$20/year

founding price, paid upfront): a station running this same kit at `yourname.gamelicense.org`, set up for you after you subscribe. Hosted stations handle key redemptions only. That is GameLicense's service, not FlipKey's: GameLicense bills you directly, FlipKey never holds your keys, and you use a hosted station at your own risk. Export your keys and move to your own install whenever you're ready. To cap how many unused keys can sit on any box, set `VAULT_MAX_UNUSED_PER_DROP` (unset means no cap).

Keys at rest — encrypted by default, KMS when you're ready

Since kit 0.3.0 every key is encrypted before it is stored. There are three settings for `KEY_ENCRYPTION`, and the kit is built so moving up a level is a config change plus one command, never a migration:

MODE	WHO USES IT	WHAT IT PROTECTS AGAINST	COST
<code>local</code> (default)	Every one-command droplet install. The installer generates a station master key into <code>.env</code> .	A copied database, a backup, a snapshot handed to a contractor: all unreadable without the master key. Root on the box holds both, which is this level's honest ceiling.	Nothing.
<code>kms</code>	Hosted stations on GameLicense.org, and any studio whose inventory is worth a targeted attack.	The master key never leaves AWS. A stolen database <i>and</i> a stolen <code>.env</code> are still worthless without a separate AWS compromise, and every decrypt is a CloudTrail line.	About \$1/month for the KMS key plus \$0.03 per 10,000 calls . The kit makes two calls per key, one at upload and one at redemption; AWS's free tier covers 20,000 calls a month, so a small studio pays the dollar.
<code>none</code>	Testing only, or a station from before 0.3.0 that has not been converted.	Nothing. A copy of the database is a copy of your keys. The Key Station page shows this in red.	Nothing, until it costs everything.

The envelope is the same at both levels: each key gets its own AES-256-GCM data key, wrapped by the master key, and the wrapped data key rides along in the `game_key` column. A key is decrypted once, in memory, at the moment of redemption. Dedup keeps working through `keys.key_hash`, an HMAC fingerprint of each key.

Moving from `local` to `kms`: create one KMS key used by nothing else and an IAM user with `kms:GenerateDataKey` + `kms:Decrypt` on that key only, then add the KMS lines to `.env`, switch the mode, and run `npm run reseal-keys` once while `LOCAL_MASTER_KEY` is still present. It rewrites every local envelope as a KMS one; delete the local key afterwards. Rows imported as plain text are converted by the same command. The server refuses to start if either mode is half-configured.

```
KEY_ENCRYPTION=kms
KMS_KEY_ARN=arn:aws:kms:us-east-2:123456789012:key/... # one key, used by nothing else
AWS_REGION=us-east-2
AWS_ACCESS_KEY_ID=... # IAM user with kms:GenerateDataKey + kms:Decrypt on
that key only
AWS_SECRET_ACCESS_KEY=...
KEY_HASH_SECRET=... # 32+ random chars (openssl rand -hex 32); never
change it later
```

The only code that touches the key column is `db/keycrypto.js`, via `insertKey` (seal) and `claimKey` (open) — swap that one module for GCP KMS, Azure Key Vault, or HashiCorp Vault. The kit's README, security checklist item 8, has the least-privilege IAM policy and the `--rebind` step for KMS stations set up before 2026-09-13.

Stock sync — FlipKey sells only what your station holds

Since kit 0.2.0 the station reports its *count* of unused keys per listing and platform to FlipKey.gg every ten minutes and whenever the count changes. Counts only, never keys, and buyers never see the number. FlipKey.gg sells a listing only while the station reports keys for it, probes the station once more in the seconds before any charge, and **stops selling after 30 minutes of silence**, so a station that is down pauses its own sales instead of selling keys it cannot hand over. Buyers who already paid are told the station is offline and offered a no-questions refund inside their window. A new listing stays off sale until its first keys land. If you run your own stack without the kit, none of this applies: your listings sell up to their quantity as before.

Upgrading your station

Your Key Station page shows an **upgrade available** notice whenever a station reports an older kit than the current one (`0.3.2` today). Nothing forces you: a station on an older kit keeps redeeming, and FlipKey.gg never refuses a redemption or a sale because of the kit version. What you give up by staying behind is listed per release in the kit README.

- **Hosted on GameLicense.org:** GameLicense upgrades it for you. If the notice stays for more than a day, email support@gamelicense.org.
- **Your own droplet:** click *Get my upgrade command* on the Key Station page, paste it in the droplet console. About a minute; your `.env`, keys and database stay where they are, and the previous code is kept beside the install.
- **Your own stack:** there is nothing to upgrade on your side unless you want a change. The Redeem SDK widget loads from flipkey.gg, so your page always runs the current widget; the handler contract it speaks is documented in Redemption SDK, changes are listed in that page's changelog, and when the signed message changed the kit kept accepting the previous shape so a station is never stranded mid-upgrade.

Release notes, newest first: **0.3.2** the optional Steam authenticity floor fails closed and the installer takes your own Steam Web API key · **0.3.1** dependency refresh · **0.3.0** keys encrypted at rest by default (`local`) · **0.2.1** browser uploads fixed · **0.2.0** stock sync and In-house listings. Older than 0.2.0 means the station never reports, so FlipKey.gg cannot tell what it runs and the page treats it as behind.

Going live — checklist

- `FLIPKEY_WEBHOOK_SECRET` matches what FlipKey.gg issued you (otherwise the link test fails with "signature mismatch").
- **Token Mirror Endpoint is set on the Webhooks page:** `https://<your-host>/api/mirror/token-mint`. Every sale POSTs a signed `purchase` event here so your `tokens` mirror learns which token belongs to which drop. Without it, buyers get "Token not registered with this publisher" at redemption. (For local testing without a public URL, `scripts/register-tokens.js` fills the mirror by hand.)

- `NFT_CONTRACT_ADDRESS` is your deployed contract, not a placeholder.
- `CHAIN_RPC_URL` is a paid endpoint (public RPCs rate-limit and fail open at scale).
- HTTPS in front of the app — wallet signing requires a secure context.
- You've imported keys for every variant on every drop you list.
- Pass FlipKey.gg's link test on each redemption URL when listing a game (the listing form runs this for you, green/red per variant).
- Keys at rest are `local` (the installer's default) or `kms` — never `none` once real keys are loaded. See "Keys at rest" for the step up to KMS and what it costs.
- Backups configured for `db/keys.db` (or your equivalent).
- Logging strips the `key` field — see the security checklist below.

Customization — plugging in your existing systems

IF YOU HAVE...	EDIT...
An existing key store (vault, encrypted DB, separate service)	<code>db/index.js</code> → replace <code>claimKey</code> with a call to your store. The redeem handler doesn't care where the key comes from.
An existing auth / session system	Wrap the <code>/api/redeem</code> route in your auth middleware. Wallet signature is the canonical proof; extra session checks are fine.
Existing logging / APM	Be explicit — if your APM auto-captures responses, redact the <code>key</code> field. The <code>redeem.js</code> route comment marks where.
MySQL or Postgres already running	<code>DB_DRIVER=mysql</code> (or <code>postgres</code>) + <code>DATABASE_URL=...</code> in <code>.env</code> . Done.

Reimplementing in your stack (the spec)

The protocol is what matters; the language is incidental. The reference implementation in `routes/redeem.js` is ~200 lines including comments — your port should be similar.

CONCEPT	IMPLEMENTATION IN THE KIT	YOUR EQUIVALENT
Endpoint shape	<code>POST /api/redeem</code> returns one of <code>{result:"redirect" "entitlement_granted" "show_key", ...}</code>	Same. The Redeem SDK is wire-protocol-compatible regardless of backend.
Cache headers	<code>Cache-Control: no-store</code> + <code>Pragma: no-cache</code> on the redeem response	Mandatory. Set unconditionally.
Signature message format	<code>routes/redeem.js</code> → <code>buildSignableMessage</code>	Match exactly. Any drift breaks verification.
Signature verification	<code>viem.verifyMessage</code> (handles EOA + ERC-1271 + ERC-6492)	Python: <code>eth-account</code> for EOA + custom ERC-1271 RPC call. Go: <code>go-ethereum/crypto</code> + ERC-1271 contract call. ERC-6492 has bespoke prefix handling.
Replay protection	<code>nonces</code> table; insert-with-unique-constraint blocks reuse	Same idea — Redis with TTL, or any KV with atomic SETNX.
Ownership check	<code>ERC1155.balanceOf(wallet, tokenId) >= 1</code> via JSON-RPC	Same. Any web3 library can call <code>eth_call</code> .
Atomic key claim	<code>UPDATE keys SET status='redeemed' WHERE id = (SELECT LIMIT 1) RETURNING ...</code>	Critical: one statement (or a transaction with a row-level lock). SELECT-then-UPDATE creates a race.
Activation report	HMAC-SHA256 the JSON body, send <code>X-FlipKey-Signature: <hex></code>	Same. The shared secret is your <code>FLIPKEY_WEBHOOK_SECRET</code> .
Link test	Match <code>?fk_test=1&nonce=...</code> return <code>{nonce, sig: hmac_sha256(secret, "fk_link_test:" + nonce)}</code>	Trivial in any framework. Mount before your real redemption page.

Security checklist

Read these. Most "we got pwned and lost a quarter of inventory" stories trace back to one of these twelve items.

- Never put the key in HTML.** No `data-key`, no `<input value=...>`, no server-rendered key in any DOM attribute. Extensions, session-replay tools, proxies, and DOM-snapshotting APMs all scrape what's there. Return the key only in the JSON response body to `/api/redeem`.
- Cache-Control: no-store** on every response that may carry key material — CDN/proxy/browser caches persist it otherwise. The starter sets this at the top of the redeem handler; do not remove it.
- Atomic key claim.** Single-statement `UPDATE ... RETURNING`. Two-statement SELECT-then-UPDATE lets two concurrent redemptions grab the same key.
- Don't log the key.** Strip `game_key` from APM, error reporters (Sentry, Bugsnag), and access logs — these often capture full responses by default.
- Don't email the key.** SMTP is plaintext, mailboxes are searchable, recovery emails leak. Deliver via the SDK's response only.
- HTTPS everywhere.** Both the redemption page and `/api/redeem`. Wallet signing requires a secure context.

7. **Replay protection.** The `nonces` table blocks signature reuse. Confirm it: POST the same nonce twice against prod — the second must 400.
8. **Encrypt keys at rest in production.** On by default since 0.3.0 (`local`): station master key in `.env`). Step up to `KEY_ENCRYPTION=kms` with your own KMS key when the inventory warrants it — see "Keys at rest" above. What goes in KMS is the *encryption* key, not the game keys: a stolen DB dump is then worthless without a separate KMS compromise. Plaintext (`none`) is for testing only.
9. **Least-privilege DB user.** The app account needs `SELECT/INSERT/UPDATE` on four tables and nothing else. Run `init-db` as admin once; run the server as the restricted user. Keep the DB reachable only from the app host.
10. **Import in waves, not the whole allocation.** Your blast radius is whatever's in the `keys` table. If a drop redeems ~2,000 keys this month, import ~2,000 — not the 50,000-key master allocation. Top up as buckets run low. The single cheapest defense: a fully compromised server can only lose what you loaded.
11. **Watch claim velocity, and know your revocation path.** Alert when redemptions-per-hour jumps far above your sales rate — that's bulk theft from the inside. If a batch leaks: flip those rows to `status='revoked'` (the claim query skips them), and for unredeemed Steam keys ask Valve to deactivate the batch. Stolen keys that can be switched off are a contained incident, not a catastrophe.
12. **Backups inherit the crown jewels.** An encrypted-at-rest DB exported to an unencrypted bucket or laptop dump is how stashes walk out. Encrypt backups with a different key than the live DB, and restrict who can read them.

What's NOT in v1

Real publisher needs we deliberately deferred — next steps, not bugs:

- **Admin dashboard** — keys-per-bucket inventory view, low-stock alerts, redemption reports. v1.1.
- **Steam revocation API integration** — fire `RemoveAppKeyForUser` via Steamworks when a key is stolen. Pattern documented; doesn't ship.
- **Hot/cold key separation** — for catalogs > 10k keys per bucket, periodic top-up from a vault into the active DB.
- **Bundle redemptions** — one NFT redeems multiple products. Rare; deferred.
- **Pre-assigned keys** — token-to-key mapping at mint time vs. the starter's lazy claim at redemption.

MIT licensed. Fork freely.

The kit's README ships inside the zip and goes deeper than this page: full release notes, the hand-upgrade steps, the IAM policy for KMS, and the thirteen-item security checklist.