LawnPro Software

Integration Guide

Step-by-step: from API key to go-live.

Public REST API v1 · https://secure.lawnprosoftware.com/api/v1
Generated August 3, 2026 · Support 205-369-7052, 7am–7pm CST

LawnPro API & Webhooks — Integration Guide

Start here. This guide walks a working integration end to end: getting a key, making your first call, building a client that survives production, receiving webhooks, and going live. It is instructions, not a catalog.

When you need field-level detail, use the reference docs instead:

Document	Use it for
`README.md`	One-page overview and cheat sheet.
`REFERENCE.md`	Every endpoint, field, filter, and enum.
`WEBHOOKS.md`	Complete webhook spec (envelope, signing, retries, event catalog).
`openapi.yaml`	Machine-readable OpenAPI 3.1 contract — generate a client from this.

- **Base URL:** `https://secure.lawnprosoftware.com/api/v1`
- **Auth:** `Authorization: Bearer lp_live_<prefix>.<secret>`
- **Everything is JSON**, both directions.

> **A note on vocabulary.** LawnPro says **customer** (not client), **property** (not job site), **visit** (not appointment or job), and **estimate** (not quote). The API field names follow that, and so does this guide. Sections marked **LawnPro-specific** describe behavior that won't match your instincts from other platforms — read those even if you skim the rest.

Table of contents

---

1. Before you start

What you need:

- A LawnPro account, and the **account owner's** cooperation — only the owner (`user_role = 1`) can request an API key or manage webhooks. A staff login with full settings access still cannot, deliberately: a key carries the whole company's data.
- Somewhere to keep a secret. The key is shown once and is equivalent to a password for the entire account.
- For webhooks: a public HTTPS URL. Not `localhost` — see Step 8.

What the API can do today:

Resource	Read	Create / update / delete
Customers	yes	yes
Properties	yes	yes
Items & Services	yes	yes
Invoices	yes	**no**
Estimates	yes	**no**
Payments	yes	**no**
Schedule / visits	yes	**no**
Webhook endpoints	yes	yes

LawnPro-specific: invoices, estimates, payments, and the schedule are **read-only in v1**. You can pull them out; you cannot create or change them through the API. If your integration needs to write an invoice, that work does not exist yet — plan around it rather than around a workaround.

Is it turned on? API access is currently a **pilot feature behind a flag**, not a plan entitlement. If **Settings → API Access** isn't visible, the account hasn't been enabled and no amount of correct code will help — contact support first. Webhooks have their own separate flag, so an account can have webhooks without API keys (and enabling API access grants webhooks too).

Step 1 — Get an API key

Keys are **self-service to request, staff-approved to issue**. Three moves:

1. **The owner requests it.** In LawnPro: **Settings → API Access → Request an API key**. Give it a label (e.g. "Acme CRM sync") and say what it's for. Be specific about what the integration does — that text is what staff use to decide the scopes.
2. **LawnPro staff approve it** and set the scopes and rate limit. The request shows as **Awaiting approval** until then, and the owner can **Cancel** it. Only **one open request at a time** per account, so don't queue up several.
3. **The owner generates the key.** Once the row shows **Approved**, the owner clicks **Generate key**.

The full key appears **exactly once**, right then.

Copy it immediately into your secrets manager. Only a public prefix and a bcrypt hash of the secret are stored, so nobody — not support, not the owner — can retrieve it later. Lost key means requesting a new one and revoking the old.

> **The secret is generated in the owner's own browser session at the moment they click Generate key.** LawnPro staff approve the *request*; they never see the resulting secret. If anyone asks you to send them a key, that is not how this works.

Ask for the right scopes

Scopes are `resource:action`, and a read scope never implies write. Ask for the narrowest set that does the job — a reporting dashboard should get `customers:read` and `invoices:read`, nothing more.

`customers:read` · `customers:write` · `properties:read` · `properties:write` · `items:read` · `items:write` · `invoices:read` · `estimates:read` · `payments:read` · `schedule:read` · `webhooks:read` · `webhooks:write`

A call missing its scope fails with `403 insufficient_scope` and names the scope it wanted, so this is quick to diagnose — but changing scopes means a new approval, so get it right at request time.

Step 2 — Make your first call

Prove the key works before you write any code:

```
curl -i "https://secure.lawnprosoftware.com/api/v1/customers?per_page=1" \
  -H "Authorization: Bearer lp_live_<prefix>.<secret>"
```

A `200` with a `data` array means you're connected. Look at the response headers too — they matter later:

```
X-Request-Id: req_2f1c9a7b          <- quote this to support
X-RateLimit-Limit: 120
X-RateLimit-Remaining: 119
```

If it fails, jump to Troubleshooting. The three usual causes are a missing `Bearer ` prefix, a truncated secret, and the account not having the feature enabled.

Never put the key in a browser, mobile app, or anything a customer can view source on. Calls are server-to-server. CORS is restricted to a short allowlist of LawnPro's own origins, so a browser fetch from your domain will be blocked anyway — treat that as a guardrail, not the reason.

Step 3 — Read the envelope correctly

Every response — success and failure alike — has the same three keys:

```
{
  "data":   { "id": 42, "first_name": "Jane", "last_name": "Doe" },
  "meta":   { "request_id": "req_2f1c9a7b" },
  "errors": []
}
```

Lists put the rows in `data` and add pagination to `meta`:

```
{
  "data": [ { "id": 42 }, { "id": 41 } ],
  "meta": {
    "request_id": "req_2f1c9a7b",
    "pagination": { "page": 1, "per_page": 50, "total": 213, "total_pages": 5 }
  },
  "errors": []
}
```

Write your client against this shape once, centrally:

```
def call(method, path, **kw):
    r = requests.request(method, f"{BASE}{path}", headers=HEADERS, timeout=30, **kw)
    body = r.json()
    if body["errors"]:
        raise LawnProError(body["errors"][0]["code"],
                           body["errors"][0]["message"],
                           body["meta"]["request_id"])
    return body["data"], body["meta"]
```

Branch on `errors[0].code`, never on `message`. Codes are a stable contract; message wording is not.

Log `meta.request_id` on every failure. It's the only thing that lets support find your exact call in the server-side audit log.

Step 4 — Handle errors like production will

Four failures will happen to you in the first week. Handle them before launch.

429 — rate limited

Limits are **per key, per minute**, in a fixed 60-second window. Default is **120 requests/minute**; support can raise it per key. You get:

```
Retry-After: 37
```

Sleep that many seconds and retry. Don't guess, don't hammer — and don't ignore `X-RateLimit-Remaining`, which lets you slow down *before* you get blocked.

503 — the limiter itself is unavailable

Rare, but real, and it surprises people: the rate limiter **fails closed**. If its counter errors, you get `503` with code `rate_limiter_unavailable` and `Retry-After: 5` rather than being let through. Treat it exactly like a `429`: back off and retry.

401 — and the brute-force throttle behind it

Bad keys get `401 unauthorized`. Note that **repeated auth failures from one IP are throttled**: past 30 failed attempts in a minute you get `429` instead of `401`. So a retry loop wrapped around a wrong key digs itself deeper. Fail fast on `401` — it will never fix itself by retrying.

422 — validation

`422 validation_failed` includes a `fields` array naming what was wrong:

```
{ "code": "validation_failed", "message": "Invalid email address.", "fields": ["email"] }
```

Surface `fields` to whoever is feeding you data. Retrying an unchanged `422` always fails again.

A retry policy that works

Status	Retry?	How
`429`, `503`	Yes	Honor `Retry-After`.
`500`	Yes	Exponential backoff, and use an `Idempotency-Key` on creates.
`401`, `403`, `404`, `405`, `413`, `422`	**No**	Fix the request or the key.
`409 idempotency_conflict`	Yes, shortly	Your own identical create is still in flight.

Full code table: `REFERENCE.md` — §6 Errors.

Step 5 — Page through lists

`page` starts at 1; `per_page` defaults to 50 and is **capped at 100** (higher values are silently clamped, not rejected). Loop until you reach `total_pages`:

```
page = 1
while True:
    data, meta = call("GET", "/customers", params={"page": page, "per_page": 100})
    for row in data:
        handle(row)
    pg = meta["pagination"]
    if pg["page"] >= pg["total_pages"]:
        break
    page += 1
```

Two practical notes. A large account is thousands of customers, so a full sync is dozens of calls — budget it against your rate limit rather than firing them all at once. And filter server-side where you can (`customer_id`, `status`, and for the schedule `from`/`to`/`type`) instead of pulling everything and filtering in your own code.

Step 6 — Write data safely

Creates and updates exist for customers, properties, and items & services.

Always send an Idempotency-Key on creates

A network timeout on a `POST` leaves you genuinely unsure whether the record was created. Send a UUID you generate:

```
curl -s -X POST https://secure.lawnprosoftware.com/api/v1/customers \
  -H "Authorization: Bearer lp_live_<prefix>.<secret>" \
  -H "Content-Type: application/json" \
  -H "Idempotency-Key: 7c2b9f04-2f1a-4e3a-9a1d-6f5e2b8c1d22" \
  -d '{
        "first_name": "Jane",
        "last_name": "Doe",
        "email": "jane@example.com",
        "address": "123 Oak St",
        "city": "Birmingham",
        "state": "AL",
        "postal_code": "35203"
      }'
```

Retry with the **same** key and you get the original response replayed — `Idempotent-Replayed: true`, no duplicate row. Use a **fresh** key for a genuinely new record. Keys are scoped per endpoint, and failed creates release the key so you can cleanly retry rather than replaying an error forever.

Updates are partial

`PUT` (and `PATCH`) change **only the fields you send**. Omitted fields are left alone, so you never need to read-modify-write a whole object:

```
curl -s -X PUT https://secure.lawnprosoftware.com/api/v1/customers/42 \
  -H "Authorization: Bearer lp_live_<prefix>.<secret>" \
  -H "Content-Type: application/json" \
  -d '{ "phone": "205-555-0199" }'
```

Unknown fields in a body are ignored, not rejected — which means a typo'd field name fails **silently**. If a write seems to do nothing, check your spelling against `REFERENCE.md` — §12 Resources.

Deletes are soft

`DELETE` returns `{ "id": 42, "deleted": true }` and soft-deletes the record. It disappears from the API; it is not destroyed in LawnPro.

Bulk imports: suppress the automations

LawnPro-specific, and the mistake is expensive. Creating a customer fires that company's "customer added" automations — which can mean a real welcome email or text to a real person. Importing 500 customers with the defaults sends 500 messages.

Pass `"trigger_automations": false` on every create in an import:

```
{ "first_name": "Jane", "last_name": "Doe", "trigger_automations": false }
```

It must be a **real JSON boolean**. The check is strict (`=== false`), so `"false"` as a string, `0`, or `null` all count as "fire the automations." Verify with one test customer before running the batch.

Other validation worth knowing up front

- A customer needs **at least one** of `company_name`, `first_name`, `last_name`.
- A property's `customer_id` must be a customer in **your** account, or you get `422`.
- An item's `category_id` must be yours; if you omit it on create, it defaults to the account's first category, and the create **fails** if no category exists yet.
- Creating customers is subject to the account's plan limit — over it, you get `403 forbidden` rather than a validation error.

Step 7 — Receive webhooks instead of polling

Polling for "did anything change" wastes your rate limit and is always a little stale. Webhooks push the event to you when it happens. Register one HTTPS endpoint per system that needs events.

Full spec: `WEBHOOKS.md`. The short version:

Register the endpoint

In the app: **Settings → Webhooks → Add endpoint** (owner only). Or via API with `webhooks:write`:

```
curl -s -X POST https://secure.lawnprosoftware.com/api/v1/webhooks \
  -H "Authorization: Bearer lp_live_<prefix>.<secret>" \
  -H "Content-Type: application/json" \
  -d '{
        "url": "https://example.com/hooks/lawnpro",
        "events": ["invoice_paid_in_full", "customer_added"],
        "description": "Sync to our CRM"
      }'
```

The response contains `data.secret` (`whsec_…`) **once**. Store it now — it's never returned again, and rotating means deleting the endpoint and creating a new one.

> **Two different secrets.** The **API key** (`lp_live_…`) authenticates your calls *to* LawnPro. The **signing secret** (`whsec_…`) verifies deliveries *from* LawnPro. They are not interchangeable. If you only use the UI, you don't need an API key for webhooks at all.

Subscribe to specific event names, or `"*"` for everything. The authoritative list is the event picker in **Settings → Webhooks**; the catalog is in `WEBHOOKS.md` — §7.

What arrives

`POST`, JSON, and you must answer within **10 seconds**:

```
{
  "id": "evt_9f8c1b2a3d4e5f60718293a4b5c6d7e8",
  "event": "invoice_paid_in_full",
  "created_at": "2026-07-11T18:03:22Z",
  "company_id": 1234,
  "object": "invoice",
  "object_id": 987,
  "data": { "id": 987, "customer_id": 42, "total": 250.00, "balance_due": 0.00 }
}
```

`data` is the affected object in the same shape the REST API returns it. If the record can't be fully loaded at send time it degrades to a minimal `{ "id": …, "customer_id": … }` — always enough to fetch the rest from the API, so your handler should tolerate the thin version.

The four rules for your receiver

Get these right and webhooks are boring, which is what you want.

1. Verify the signature over the raw body. The HMAC covers `"<timestamp>.<raw body>"`. If you parse the JSON and re-serialize it before hashing, whitespace differences will break the comparison and you'll chase it for an afternoon.

```
$payload   = file_get_contents('php://input');          // raw bytes, unparsed
$timestamp = $_SERVER['HTTP_X_LAWNPRO_TIMESTAMP'] ?? '';
$sigHeader = $_SERVER['HTTP_X_LAWNPRO_SIGNATURE'] ?? '';
$expected  = 'sha256=' . hash_hmac('sha256', $timestamp . '.' . $payload, $YOUR_SECRET);

if (!hash_equals($expected, $sigHeader) || abs(time() - (int) $timestamp) > 300) {
    http_response_code(400);
    exit('invalid signature');
}
```

Compare in **constant time** (`hash_equals`, `crypto.timingSafeEqual`, `hmac.compare_digest`) and reject timestamps more than 300 seconds old. Node and Python versions: `WEBHOOKS.md` — §4.

2. Acknowledge fast, work later. Return any `2xx` within 10 seconds. Queue the actual processing. If you do the CRM sync inline and it takes 12 seconds, LawnPro treats it as failed and retries — and now you're processing twice.

3. De-duplicate on `X-LawnPro-Delivery`. Delivery is **at least once**. The same event can arrive twice (your `2xx` got lost, a worker died after sending). That header is stable across retries of the same delivery, so record it and skip what you've seen.

4. Don't rely on order, and ignore unknown events. Deliveries are best-effort in order but **not guaranteed**, so key decisions off the object's own state or `created_at`, not arrival sequence. And tolerate event names you didn't subscribe to — the `ping` test event is delivered regardless of your subscription list.

When you fail

Non-2xx, timeout, connection error, or DNS failure all count as failure. LawnPro retries with backoff — 1 minute, 5 minutes, 30 minutes, 2 hours, 6 hours — for **6 attempts total** (the first plus five retries), about 8½ hours end to end. After the sixth the delivery is dead-lettered and never retried.

Watch `consecutive_failures` on the endpoint, or `GET /api/v1/webhooks/{id}/deliveries` for the last 20 attempts with the real HTTP status and error. An endpoint quietly failing for nine hours has already lost data.

Step 8 — Test your webhook receiver

Send a real test event. In **Settings → Webhooks**, the paper-plane button queues a `ping` through the actual pipeline — same signing secret, same SSRF checks, same delivery log. A `ping` that shows **delivered** is genuine proof, not a simulation. Owner only, active endpoints only, one per endpoint per minute, and **one attempt with no retries**, so a failed test stays failed. Allow up to a minute for the worker to pick it up.

Because it's really signed, `ping` is the cheapest way to exercise your signature-verification code before live data flows.

You cannot point an endpoint at `localhost`. The worker re-resolves the hostname at delivery time and refuses anything resolving to a private, loopback, or reserved address — so tunnel your dev machine (ngrok or similar) and register the HTTPS tunnel URL. While you're still sketching, a request bin like webhook.site

shows you full headers and body, and Svix Play will check an HMAC for you.

Go-live checklist

Your API client

- ☐ Key lives in a secrets manager or env var — never in source control, never client-side.
- ☐ Honors `Retry-After` on `429` **and** `503`.
- ☐ Does not retry `401`/`403`/`422` (and knows repeated `401`s get throttled).
- ☐ Sends `Idempotency-Key` on every create.
- ☐ Pages to `total_pages` rather than assuming one page.
- ☐ Logs `meta.request_id` on failures.
- ☐ Branches on `errors[0].code`, not on message text.
- ☐ Bulk imports send `"trigger_automations": false`.

Your webhook receiver

- ☐ Public HTTPS.
- ☐ Verifies the signature over the **raw** body, constant-time, with the 300-second window.
- ☐ Returns `2xx` in under 10 seconds; heavy work is queued.
- ☐ Idempotent, keyed on `X-LawnPro-Delivery`.
- ☐ Survives out-of-order and duplicate deliveries.
- ☐ Ignores unknown event names, including `ping`.
- ☐ `whsec_` secret stored at creation.
- ☐ Someone is alerted on `consecutive_failures` climbing.

Operationally

- ☐ A `ping` has been delivered successfully to the production URL.
- ☐ You know who to call: **205-369-7052**, 7am–7pm CST, 7 days.
- ☐ If you're migrating off the legacy `/api`, that surface is deprecated — see `REFERENCE.md` — §16.

Troubleshooting

What you see	Why	Fix
`401 unauthorized` on every call	Missing `Bearer ` prefix, truncated secret, or key was revoked	Re-check the header format; confirm the key shows **Active** in Settings → API Access
`429` immediately, with no traffic	Failed-auth throttle (30 bad auths/IP/minute)	Stop retrying, fix the key, wait for the window
`403 insufficient_scope`	Key wasn't granted that scope	New request with the right scopes — scopes aren't editable after issuance
`403 forbidden`	Account suspended, or over the plan's customer limit on a create	Check account standing / plan

What you see	Why	Fix
`403 https_required`	Called over plain HTTP	Use `https://`
`404` on a record you can see in the app	The record belongs to another company, or is soft-deleted	Confirm the ID came from this same key's account
`405 method_not_allowed`	Writing to a read-only resource (invoice, estimate, payment, schedule)	Not supported in v1
`422` naming `customer_id`	Referencing another company's customer	Use an ID from your own account
Write returns `200` but nothing changed	Misspelled field name — unknown fields are ignored silently	Check spelling against `REFERENCE.md` §12
Settings → API Access isn't there	Account isn't in the pilot	Contact support
Signature never verifies	Hashing re-serialized JSON instead of the raw body	Capture raw bytes before parsing
Webhook retried even though you returned 200	Took longer than 10s to answer	Acknowledge first, queue the work
Same event handled twice	Normal — delivery is at least once	De-duplicate on `X-LawnPro-Delivery`
Test event never arrives	Endpoint paused, URL not public, or under the 1/minute throttle	Check **Recent deliveries** for the real status code

When you contact support about a specific call, bring the `request_id`. It turns "the API is broken" into a one-minute lookup.

LawnPro-specific gotchas

Collected in one place, because these are what cost outside developers a day:

1. **Half the API is read-only.** Invoices, estimates, payments, and schedule cannot be written in v1.
2. **Creating a customer can send a real message.** Automations fire on create. Use `"trigger_automations": false` for imports.
3. **Statuses are integers, not strings.** `custom_status = 1` is active, `6` is a lead; invoice `2` is paid, `4` past due. Tables: `REFERENCE.md` — §14 Enumerations.
4. **`pay_method = 4` is not money.** Payment method 4 is "Customer Credit Balance," an internal credit application. Exclude it from revenue math.
5. **Access is a feature flag, not a plan tier.** API access is a pilot; webhooks have their own flag. Neither is something a merchant can switch on themselves.
6. **Only the account owner** can request keys or manage webhooks.
7. **The rate limiter fails closed** — expect `503` as well as `429`.
8. **Unknown request fields are ignored silently.** Typos don't error.
9. **Deletes are soft**, everywhere.
10. **The two customer-scoring events ignore `"*"`.** `customer_score_at_risk` and `customer_churn_risk_flagged` must be named explicitly (they're Plus-only and model-driven, not user actions), e.g. `["*", "customer_score_at_risk"]`.