



Partner Integration Manual

Connecting your application to IPS pool and spa chemical controllers

IPS Controllers Partner API v2.0

Edition 1.0 · October 2026

API reference: <https://integrations.ipscontrollers.com/openapi.yaml>

Contents

1	How the integration works	9	Errors and rate limits
2	Getting started	10	Security requirements
3	Addresses and environments	11	AI agents and code generation
4	Connecting a customer (OAuth 2.1)	12	Going live
5	Permissions	A	Endpoint reference
6	Reading data	B	Data reference
7	Changing settings	C	Glossary
8	Webhooks		

About this manual

This manual explains how your application connects to IPS Controllers on behalf of IPS customers: how a customer approves your application, what you can read, how settings changes work, how events reach you, and what IPS requires of you before going live.

The **OpenAPI 3.1 document** is the authoritative reference for every endpoint, field and event. Generate clients from it, and give it to AI coding agents (chapter 11). Where this manual and the OpenAPI document differ, the OpenAPI document is correct.

Your **IPS partner contact** issues your credentials and demo accounts, answers questions, and approves you for production.

Already using the original IPS API?

The original IPS API at <https://monitor.ipscontrollers.com/api/> keeps working exactly as it does today. Version 2 is IPS's partner platform from here on: customers approve access themselves, permissions are enforced on every call, and events arrive as signed webhooks. Move to it at your own pace, with your IPS partner contact.

1 How the integration works

IPS controllers measure and dose pool and spa chemistry: pH, sanitizer (ORP), temperature and, on some models, free chlorine. The Partner API lets your application work with the controllers a customer chooses to share with you, and only those.



Your application talks only to the IPS Partner Platform. IPS talks to the controllers.

Four principles

The customer decides

The customer signs in to IPS (never to you), picks which controllers to share, and chooses per controller **View only** or **View and change settings**. That approval is a **grant**.

Checked on every call

Your token carries no permissions. IPS checks the grant on every request, so a customer who narrows or revokes access is obeyed on your very next call.

Settings are requests

These controllers dose chemicals. A settings change is accepted as a request (202), reaches the controller on its next check-in, and its outcome arrives as an event.

Events come to you

Alarms, offline and online status, and settings outcomes are pushed to your HTTPS endpoint as signed webhooks, so you don't need to poll.

What partners build

- **Service and route software:** show each customer's pools with live readings and alarms, and dispatch a technician when an alarm is raised.
- **Building and property systems:** bring pool status into one dashboard, and alert on flow, tank and chemistry alarms.
- **Automation tools:** trigger workflows from controller events, or adjust set points within the limits IPS enforces.

Partners can never create IPS accounts, add controllers, or see a customer's password. Customers add controllers in the IPS app, as they do today.

2 Getting started

1. **Talk to your IPS partner contact.** Describe your integration: what you need to read, whether you need to change settings, and whether you want webhooks. IPS approves your integration for some or all of three capabilities: `read`, `settings_write` and `webhooks`.
2. **Send your redirect URIs.** These are the exact addresses IPS returns customers to after they approve, for example `https://app.example.com/ips/callback`. They must be HTTPS. `http://localhost` is allowed for development clients only.
3. **Receive a demo client and demo accounts.** IPS issues a `client_id`, a `client_secret` (shown to IPS once and passed to you securely), and the sign-ins of demo customers with simulated controllers.
4. **Build and test against the demo accounts** (chapters 4 to 9).
5. **Complete the go-live checklist** (chapter 12). IPS then issues your production client.

Keep the client secret on your server.

It identifies your company. Never put it in a mobile app, a browser or a public repository. If it is exposed, ask IPS to rotate it.

What you receive

Item	What it is
<code>client_id</code>	Public identifier of your application.
<code>client_secret</code>	Your application's password for the token endpoint. Server-side only.
Redirect URIs	Exactly as registered. No wildcards. Any change goes through IPS.
Capabilities	Which of <code>read</code> , <code>settings_write</code> , <code>webhooks</code> you may request as OAuth scopes. Customers are never offered a choice your integration can't use.
Demo accounts	Demo customers at <code>@demo.ipscontrollers.com</code> . A demo client can be connected by demo accounts only.

3 Addresses and environments

Purpose	Address
API base	<code>https://integrations.ipscontrollers.com/v2</code>
Authorization (send the customer here)	<code>https://integrations.ipscontrollers.com/oauth/authorize</code>
Token	<code>https://integrations.ipscontrollers.com/oauth/token</code>
Token revocation	<code>https://integrations.ipscontrollers.com/oauth/revoke</code>
Server metadata (RFC 8414)	<code>https://integrations.ipscontrollers.com/.well-known/oauth-authorization-server</code>
OpenAPI document	<code>https://integrations.ipscontrollers.com/openapi.yaml</code> and <code>/openapi.json</code>
Documentation	<code>https://integrations.ipscontrollers.com/docs</code>

Read the endpoints from the server metadata rather than hard-coding them, where your OAuth library supports it.

Demo and production

One host serves both. What differs is the client you use:

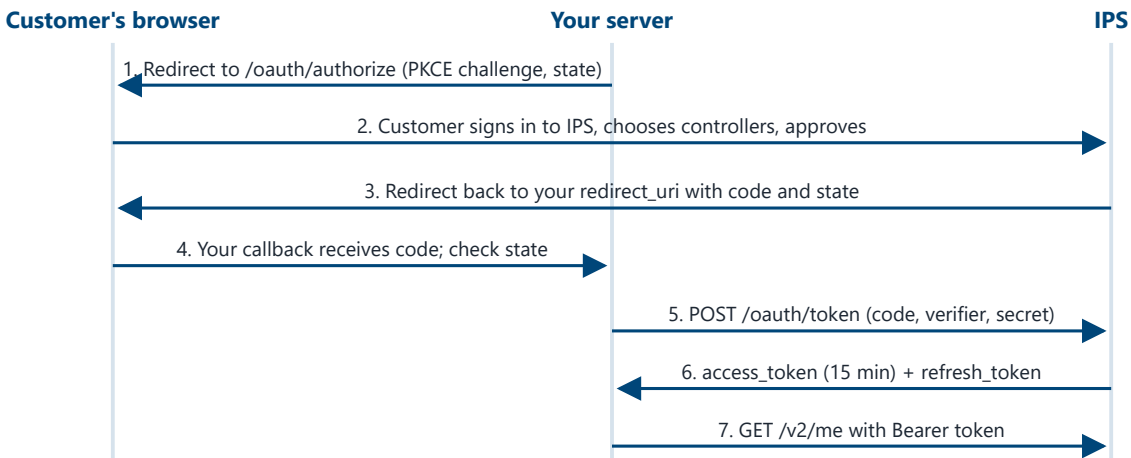
	Demo client	Production client
Who can connect it	Demo accounts (<code>@demo.ipscontrollers.com</code>) only	Real IPS customers
Controllers	Simulated, with live behaviour	Real equipment
API, scopes, events	Identical	

How the simulated controllers behave

- They report every five minutes, with three days of history from the start.
- An alarm is raised now and then and clears itself after about ten minutes.
- On request, IPS can take a demo controller offline or raise a test alarm.
- A settings change is picked up within about a minute and reported as `settings.applied`.
- Webhooks arrive within about a minute of the event.

4 Connecting a customer (OAuth 2.1)

Customers connect through the **authorization code flow with PKCE**. PKCE (S256) and `state` are required for every client, including server-side ones. There is no client-credentials flow: every access token acts for one customer's grant.



Step 1: send the customer to IPS

Create a random `code_verifier` (43 to 128 characters) and keep it with the customer's session. Derive `code_challenge = BASE64URL(SHA256(code_verifier))`, then redirect the customer:

```
https://integrations.ipscontrollers.com/oauth/authorize
?response_type=code
&client_id=YOUR_CLIENT_ID
&redirect_uri=https%3A%2F%2Fapp.example.com%2Fips%2Fcallback
&scope=read%20webhooks
&state=RANDOM_PER_REQUEST
&code_challenge=CODE_CHALLENGE
&code_challenge_method=S256
```

Ask only for the scopes you need. The customer then sees IPS-branded pages: they sign in with their existing IPS account, see your application's name and what it is asking for, and tick the controllers to share. Nothing is ticked in advance, and settings access is never pre-selected.

Step 2: handle the callback

IPS redirects to your `redirect_uri` with `code` and `state`. Reject the request if `state` does not match what you sent. If the customer declined, you receive `error=access_denied` instead.

Step 3: exchange the code

```
POST https://integrations.ipscontrollers.com/oauth/token
Content-Type: application/x-www-form-urlencoded

grant_type=authorization_code
&code=THE_CODE
&redirect_uri=https%3A%2F%2Fapp.example.com%2Fips%2Fcallback
&code_verifier=THE_CODE_VERIFIER
&client_id=YOUR_CLIENT_ID
&client_secret=YOUR_CLIENT_SECRET
```

The response holds an `access_token` (valid 15 minutes), a `refresh_token` and the granted `scope`.

Example: the PKCE pair (Node.js)

```
import crypto from "node:crypto";

const codeVerifier = crypto.randomBytes(48).toString("base64url"); // 64 characters
const codeChallenge = crypto.createHash("sha256").update(codeVerifier).digest("base64url");
const state = crypto.randomBytes(24).toString("base64url");
// Store codeVerifier and state with the user's session, then redirect.
```

Step 4: call the API

```
GET https://integrations.ipscontrollers.com/v2/me
Authorization: Bearer ACCESS_TOKEN
```

Refreshing tokens

```
POST /oauth/token
grant_type=refresh_token&refresh_token=THE_REFRESH_TOKEN
&client_id=YOUR_CLIENT_ID&client_secret=YOUR_CLIENT_SECRET
```

Refresh tokens rotate.

Every refresh returns a new refresh token, and the old one stops working. If an old refresh token is ever presented again, IPS treats it as stolen and **revokes the customer's whole grant**. Save the new refresh token before you use the new access token, and never refresh the same token from two processes at once. A refresh token not used for 90 days expires, and the customer must connect again.

Revoking

If the customer disconnects inside your product, call `POST /oauth/revoke` with the refresh token, so the grant ends at IPS too.

5 Permissions

Your effective permission on each controller is worked out **live, on every request**, from three things:

Input	Effect
What the customer shared	Per controller: View only or View and change settings .
The customer's own role on the controller	A customer can never give you more than they have. If their role is later reduced or removed, your access shrinks with it.
Your approved capabilities	Without <code>settings_write</code> , you never get settings access, whatever was shared.

The result is `view` or `view_and_settings`, shown on each controller in `GET /me` and `GET /controllers`. Only a controller's **Owner** or **Admin** can share it.

What it means for your code

- **404** means the controller is not in this grant, or no longer is. It answers exactly as a non-existent controller does. Don't treat a 404 on a controller you knew as a bug: the customer may have stopped sharing it.
- **403** means the controller is shared, but not for what you tried, such as a settings change on a view-only share.
- Don't cache permissions for long. When you get a 403 or 404 you didn't expect, read `GET /me` again.
- Customers manage what they share in IPS, under **Connected apps**: they can add or remove controllers, change the permission, or disconnect you. You don't need to send them back through authorization for that.

6 Reading data

Call	Returns
<code>GET /me</code>	The grant: the customer's display name, your scopes, and each shared controller with your permission on it.
<code>GET /controllers</code>	The shared controllers: name, model, location, MAC (for display), <code>online</code> , <code>last_report_at</code> .
<code>GET /controllers/{id}</code>	One controller's status.
<code>GET /controllers/{id}/readings? from=&to=</code>	pH, ORP, temperature, PPM, flow and tank states, newest first, over a window of up to 31 days.
<code>GET /controllers/{id}/alarms? state=open cleared all</code>	Open alarms first, then recently cleared ones.
<code>GET /controllers/{id}/settings</code>	The settings the controller last reported, and any pending change.

- **Controller IDs** look like `ctl_7Hq2mV9xLp3sK8dR`. They are opaque and stable: store them as they are. They are never a database ID or a MAC address.
- **MAC** is for display only, for example to help a technician find the right box. Never use it as an identifier.
- **Times** are UTC, RFC 3339.
- A measurement a controller model does not have is `null`. For example, a VP200 reports no temperature, and PPM appears only on PPM-capable models.
- Controllers report every few minutes, so polling readings more often returns nothing new. For alarms and status, use webhooks.

Pagination

List calls return `{ "data": [ ... ], "next_cursor": "..." }`. Pass `next_cursor` back as `cursor` until it is `null`. `limit` defaults to 100 and can be at most 500.

```
GET /v2/controllers/ctl_7Hq2mV9xLp3sK8dR/readings?from=2026-10-20T00:00:00Z&to=2026-10-21T00:00:00Z

{ "data": [
  { "controller_id": "ctl_7Hq2mV9xLp3sK8dR", "time": "2026-10-20T23:56:10Z",
    "ph": 7.4, "orp_mv": 720, "temperature": 82, "temperature_unit": "F",
    "ppm": null, "flow": "Flow OK", "tank1": "Normal", "tank2": "Normal" }
],
  "next_cursor": "eyJ0IjoimjAyNi0xMC0yMFQyMzo1MT0xMFoifQ" }
```

Alarms

Alarm `type` is one of: `no_communication`, `no_flow`, `ph_high_low`, `ph_over_timer`, `orp_high_low`, `orp_over_timer`, `tank1`, `tank2`, `tank3`, `temperature_out_of_range`, `settings_changed`, or `other` (with

`message` describing it). New types may be added: treat any you don't know as `other`. Cleared-alarm history covers each controller's recent alarms, up to about the last 25.

7 Changing settings

These controllers dose pool chemicals.

A change is never applied instantly, and your application must never change settings in a loop or retry a change automatically. Make a change only when a person asked for it, or within rules the customer configured in your product.

1. PATCH `/controllers/{id}/settings` with **only the values you want to change**:

```
{ "ph": { "setpoint": 7.5, "high_alert": 7.8 } }
```

2. IPS validates the result against the limits the IPS app enforces. Out-of-range values return **422**, with each failing field and its allowed values in `error.detail`.
3. Valid changes return **202**: `{ "change_id": "chg_...", "status": "pending" }`.
4. The controller picks the change up on its next check-in. You then receive `settings.applied` or `settings.failed`, or you can read `GET /settings-changes/{change_id}`.

Only one change per controller can be pending at a time: a second request returns **409**. You need the `settings_write` scope, **and** the customer must have shared the controller as "View and change settings".

Limits

Channel	setpoint	low_alert	high_alert
ph	7.0 to 8.0, step 0.1	6.8 to setpoint – 0.2	setpoint + 0.2 to 8.4
orp (mV)	400 to 900, step 5	100 to setpoint – 20	setpoint + 20 to 900
orp2 (mV)	390 to (orp setpoint – 10), step 5	none: follows orp	

Field	Allowed values
dose	"Continuous", "Off", or a feed time (exact string). VP200: 10, 15, 20, 30, 45, 60, 90, 120, 150, 180, 210, 240, 270, 300, 360, 480, 540, 600, 750, 900 sec (written "30 sec"). Other models: ".6 sec", "1.2 sec", "2.4 sec", "4.8 sec", "6.0 sec", "7.2 sec", "8.4 sec", "10.0 sec", "15.0 sec", "20.0 sec", "30.0 sec", "45.0 sec", "1.0 min", "2.0 min", "3.0 min", "4.0 min", "5.0 min", "8.0 min", "10.0 min", "15.0 min".
delay	Whole number 0 to 99.
overfeed	Timed dose: "N Cycles", N 0 to 100 in steps of 5. Continuous: "N Min", N 0 to 180 in steps of 5. Off: "Off".
temperature	min ≤ max ; 0 to 150 °F, or –18 to 66 °C on a controller that reports Celsius.

Rules are checked on the result (your values merged over the current ones), so moving a setpoint can mean moving its alerts in the same request.

8 Webhooks

Create a subscription with `POST /webhooks`, giving a public HTTPS `url` and the `event_types` you want. The response includes the subscription's `signing_secret`, **shown once only**. Store it securely. Send a signed test event to your endpoint at any time with `POST /webhooks/{id}/test`.

Event	When
<code>alarm.raised</code> / <code>alarm.cleared</code>	An alarm opens or clears on a shared controller.
<code>controller.offline</code> / <code>controller.online</code>	A shared controller stops reporting, or reports again.
<code>settings.applied</code> / <code>settings.failed</code>	The outcome of a change you requested.
<code>grant.revoked</code>	The customer disconnected your application. Its tokens stop working.
<code>reading.created</code>	Every reading. High volume and off by default: subscribe only if you need it.
<code>webhook.test</code>	Sent by <code>POST /webhooks/{id}/test</code> .

You receive an event only for a controller that is in the customer's grant **at the time of delivery**.

```
POST https://app.example.com/ips/events
webhook-id: evt_3Rt8yU2i06pA1sD4
webhook-timestamp: 1761170400
webhook-signature: v1,K5oZfzN95Z9UVu1EsfQmfVNQhmkZ2pj9o9NDN/H/pI4=

{ "id": "evt_3Rt8yU2i06pA1sD4", "type": "alarm.raised",
  "occurred_at": "2026-10-22T22:00:00Z", "controller_id": "ctl_7Hq2mV9xLp3sK8dR",
  "data": { "id": "alm_9Pl3kJ7hG1fD5sA2", "controller_id": "ctl_7Hq2mV9xLp3sK8dR",
    "type": "ph_high_low", "raised_at": "2026-10-22T22:00:00Z",
    "cleared_at": null, "message": "pH 7.9 is above the high alert 7.8." } }
```

Delivery and retries

- Answer with any **2xx within 10 seconds**, and do the real work after you respond.
- Failed deliveries are retried after 10 s, 1 min, 5 min, 30 min, 1 h, 2 h, 4 h, 6 h and 8 h, then every 10 h, for 24 hours in all. After that a delivery is kept, and IPS can replay it once your endpoint is fixed. A subscription whose deliveries keep failing is marked `unhealthy`, and IPS contacts you.
- Each subscription has its own delivery queue, so a slow endpoint never delays anyone else.
- **Deliveries can repeat.** The event `id` (also the `webhook-id` header) is the same on every retry: use it to ignore duplicates. Order is not guaranteed, so use `occurred_at`.

Verifying signatures (required)

Deliveries follow the **Standard Webhooks** specification. To verify one:

1. Drop the `whsec_` prefix from your secret and base64-decode the rest. That is the key.

2. Build the signed content `{webhook-id}.{webhook-timestamp}.{raw body}` from the **raw request body bytes**, not re-serialized JSON.
3. Compute HMAC-SHA256 with the key and base64-encode it.
4. Accept the delivery if it matches any `v1`, value in `webhook-signature`, compared in constant time.
Several values are sent while a secret is being rotated.
5. Reject it if `webhook-timestamp` is more than 5 minutes from your clock.

Node.js

```
import crypto from "node:crypto";

export function verifyIpsWebhook(secret, headers, rawBody) {
  const id = headers["webhook-id"];
  const ts = headers["webhook-timestamp"];
  const sigHeader = headers["webhook-signature"];
  if (!id || !ts || !sigHeader) return false;
  if (Math.abs(Date.now() / 1000 - Number(ts)) > 300) return false;

  const key = Buffer.from(secret.replace(/^whsec_/, ""), "base64");
  const expected = crypto.createHmac("sha256", key).update(`${id}.${ts}.${rawBody}`).digest();

  return sigHeader.split(" ").some((part) => {
    const [version, sig] = part.split(",");
    if (version !== "v1" || !sig) return false;
    const given = Buffer.from(sig, "base64");
    return given.length === expected.length && crypto.timingSafeEqual(given, expected);
  });
}

// Express: app.post("/ips/events", express.raw({ type: "application/json" })), ...)
// then pass req.body.toString("utf8") as rawBody.
```

Python

```
import base64, hashlib, hmac, time

def verify_ips_webhook(secret: str, headers: dict, raw_body: bytes) -> bool:
    msg_id = headers.get("webhook-id")
    ts = headers.get("webhook-timestamp")
    sig_header = headers.get("webhook-signature")
    if not (msg_id and ts and sig_header):
        return False
    if abs(time.time() - int(ts)) > 300:
        return False
    key = base64.b64decode(secret.removeprefix("whsec_"))
    signed = f"{msg_id}.{ts}.".encode() + raw_body
    expected = base64.b64encode(hmac.new(key, signed, hashlib.sha256).digest()).decode()
    for part in sig_header.split(" "):
        version, _, sig = part.partition(",")
        if version == "v1" and hmac.compare_digest(sig, expected):
            return True
    return False
```

The official Standard Webhooks libraries (`standardwebhooks` on npm and PyPI) verify these deliveries too.

9 Errors and rate limits

Every error has one shape. `message` is written for people; branch your code on `code`.

```
{ "error": { "code": "insufficient_permission",
  "message": "The customer has shared this controller for viewing only. They can allow
settings changes in IPS, under Connected apps." } }
```

HTTP	<code>code</code>	What to do
400	<code>invalid_request</code>	Fix the request; <code>message</code> says what is wrong.
401	<code>invalid_token</code>	Refresh the token. If that fails, send the customer through authorization again.
403	<code>insufficient_permission</code>	The grant does not allow it. Don't retry. The customer can widen access in IPS.
404	<code>not_found</code>	Not in this grant, or does not exist. Don't retry.
409	<code>conflict</code>	A settings change is already pending for this controller. Wait for its outcome.
422	<code>validation_failed</code>	Values out of range; <code>detail</code> lists each field.
429	<code>rate_limited</code>	Wait <code>Retry-After</code> seconds.
5xx	<code>server_error</code>	Retry reads with back-off. Don't blindly retry a settings change: check its status first.

Rate limits

Limit	Value
Per client (your application)	600 requests per minute
Per customer grant	120 requests per minute

Windows are fixed one-minute periods. Every response carries `RateLimit-Limit`, `RateLimit-Remaining` and `RateLimit-Reset` (seconds). Because the limits are per grant as well as per application, one busy customer never uses up another's allowance.

10 Security requirements

IPS customers trust you with equipment that doses chemicals. These are conditions of access:

- **Never ask a customer for their IPS password**, and never collect it. Customers sign in only on IPS pages.
- **Keep the client secret, refresh tokens and webhook secrets server-side**, encrypted at rest and out of logs. Tell IPS at once if any is exposed.
- **Verify every webhook signature** and the timestamp before acting on a delivery.
- **Use HTTPS** for redirect URIs and webhook endpoints, with valid certificates.
- **Respect revocation.** On `grant.revoked`, or a 401 your refresh can't fix, stop using that customer's tokens. Remove their data as your own terms and privacy policy require.
- **Change settings only on a person's instruction**, or within rules the customer set in your product. Show the customer what you changed and when.
- **Request the least you need:** only the scopes and event types your product uses.

11 AI agents and code generation

The OpenAPI 3.1 document is written so that AI coding agents and client generators can use it unaided. Give your agent the URL:

```
https://integrations.ipscontrollers.com/openapi.yaml  
https://integrations.ipscontrollers.com/openapi.json
```

It carries every operation, schema, error, event payload and settings limit, with a short list of rules at the top. In brief:

1. Authorization code with PKCE and `state`; every token acts for one customer.
2. Call `GET /me` first, and use the controller IDs it returns.
3. 401: refresh or reconnect. 403 and 404: don't retry.
4. Settings changes are asynchronous and must never be retried automatically.
5. Follow `next_cursor` until it is `null`; readings windows are at most 31 days.
6. On 429, wait `Retry-After`.
7. Verify webhook signatures; deduplicate on the event `id`.

A plain-text summary for AI tools is also published at <https://integrations.ipscontrollers.com/llms.txt>.

Generated code still needs your review.

Pay particular attention to settings changes: an agent must not be able to change a customer's chemistry without a person's instruction.

12 Going live

When your integration works against the demo accounts, go through this list with your IPS partner contact. IPS then issues your production client.

- ☐ The authorization flow uses PKCE (S256) and checks `state`.
- ☐ The client secret and tokens are held server-side only, encrypted, and never logged.
- ☐ Refresh-token rotation is handled: the new token is saved before use, and there are no parallel refreshes.
- ☐ 401, 403 and 404 are handled as chapter 5 describes; there are no retry loops on 403 or 404.
- ☐ Pagination follows `next_cursor`; readings requests stay within 31 days.
- ☐ Settings changes are made only on a person's instruction, never retried automatically, and their outcome is shown to the user.
- ☐ The webhook endpoint answers 2xx within 10 seconds, verifies signatures and timestamps, and deduplicates on `id`.
- ☐ `grant.revoked` stops all use of that customer's access.
- ☐ Rate-limit headers and 429 are respected.
- ☐ Production redirect URIs and webhook URLs are HTTPS and registered with IPS.
- ☐ You have told IPS who to contact about your integration, and how.

A Endpoint reference

All paths are relative to `https://integrations.ipscontrollers.com/v2`. The OpenAPI document has full request and response definitions.

Method	Path	Scope	Purpose
GET	<code>/me</code>	any	The grant and your permission on each controller
GET	<code>/controllers</code>	read	Shared controllers
GET	<code>/controllers/{controller_id}</code>	read	One controller's status
GET	<code>/controllers/{controller_id}/readings</code>	read	Readings in a window (≤ 31 days)
GET	<code>/controllers/{controller_id}/alarms</code>	read	Open and recent alarms
GET	<code>/controllers/{controller_id}/settings</code>	read	Current settings, pending change
PATCH	<code>/controllers/{controller_id}/settings</code>	settings_write	Request a change (202)
GET	<code>/settings-changes/{change_id}</code>	settings_write	Status of a change
GET	<code>/webhooks</code>	webhooks	Your subscriptions
POST	<code>/webhooks</code>	webhooks	Subscribe (secret shown once)
GET	<code>/webhooks/{webhook_id}</code>	webhooks	One subscription and its health
PATCH	<code>/webhooks/{webhook_id}</code>	webhooks	Change URL, events, or pause
DELETE	<code>/webhooks/{webhook_id}</code>	webhooks	Remove a subscription
POST	<code>/webhooks/{webhook_id}/test</code>	webhooks	Send a signed test event

OAuth endpoints

Method	Path (on the host, not under /v2)	Purpose
GET	<code>/oauth/authorize</code>	Customer sign-in and consent
POST	<code>/oauth/token</code>	Code and refresh-token exchange
POST	<code>/oauth/revoke</code>	Revoke a token
GET	<code>/.well-known/oauth-authorization-server</code>	Server metadata (RFC 8414)

B Data reference

Controller

Field	Meaning
<code>id</code>	Opaque, stable ID (<code>ctl_...</code>).
<code>name</code>	The customer's name for it, for example "Main pool".
<code>model</code>	IPS model, for example <code>M920</code> , <code>VP200</code> .
<code>location_name</code>	The customer's location, or <code>null</code> .
<code>mac</code>	Display only.
<code>online</code>	True when it has reported within its expected interval.
<code>last_report_at</code>	Time of the last report, or <code>null</code> .
<code>permission</code>	<code>view</code> or <code>view_and_settings</code> , as of this call.

Reading

Field	Meaning
<code>time</code>	When the controller took the reading (UTC).
<code>ph</code>	pH, or <code>null</code> .
<code>orp_mv</code>	Sanitizer ORP in millivolts, or <code>null</code> .
<code>temperature</code> , <code>temperature_unit</code>	Water temperature and <code>F</code> or <code>C</code> ; <code>null</code> where not measured.
<code>ppm</code>	Free chlorine or bromine on PPM-capable models, otherwise <code>null</code> .
<code>flow</code> , <code>tank1</code> , <code>tank2</code>	Flow switch and chemical tank states, as the controller reports them.

Settings change status

`pending` until the controller picks it up, then `applied` or `failed` (with `failure_reason`). `expired` is reserved for a change the controller never picks up.

Compatibility

Within v2, IPS may add fields, alarm types and event types. Your code should ignore fields and event types it does not know, and treat unknown alarm types as `other`.

C Glossary

Customer	An IPS account holder who owns or manages controllers.
Controller	An IPS chemical controller on a pool, spa or water feature.
Grant	A customer's approval of your application: which controllers, and per controller View only or View and change settings.
Effective permission	What you may do on a controller now: the grant, limited by the customer's role and your capabilities.
Capability	What IPS approved your integration for: <code>read</code> , <code>settings_write</code> , <code>webhooks</code> . Requested as OAuth scopes.
Connected apps	The page in IPS where customers see, edit and disconnect the applications they've approved.
ORP	Oxidation-reduction potential (mV), the sanitizer measurement.
Set point	The target value a controller doses towards.
Overfeed	Protection that stops dosing after a number of cycles or minutes without reaching the set point.
PKCE	Proof Key for Code Exchange (RFC 7636), which binds an authorization code to the session that asked for it.