Learn EV Charging

OCPI · VERSION 2.2.1

# The OCPI 2.2.1 Field Guide

The roles, modules, handshake, and worked examples an EV-charging integration engineer actually needs — on one desk reference.

A practitioner's reference to the Open Charge Point Interface — the protocol that lets CPOs and eMSPs roam.

learnevcharging.com

FIELD GUIDE

# What's inside

OCPI is deceptively simple to read and easy to get subtly wrong in production. This guide is the map plus the traps — written to sit next to your editor while you build.


| | | |
|---|---|---|
| 01 | **The mental model** | Roles, Sender/Receiver, and who owns what |
| 02 | **The module map** | All 12 modules at a glance |
| 03 | **The registration handshake** | Versions + Credentials, the three-token dance |
| 04 | **Auth, addressing & the response envelope** | The plumbing every call shares |
| 05 | **Worked example — Locations** | A full object, and why Connectors have no status |
| 06 | **Worked example — Sessions & Tariffs** | Live session + the step_size units trap |
| 07 | **Push vs Pull** | GET, PUT, and PATCH the right way |
| 08 | **The top 10 gotchas** | The bugs that reach production |
| 09 | **One-page cheat sheet** | Endpoints, methods, status codes |


**HOW TO USE THIS**

Every object and status code here matches OCPI 2.2.1. Where 2.2.1 differs from 2.1.1 or 2.3.0, it's flagged. This is a reference, not a spec replacement — the full spec lives at evroaming.org.

01 · THE MENTAL MODEL

# Roles, Sender/Receiver, and who owns what

OCPI (Open Charge Point Interface) is the roaming protocol that lets a driver with one provider charge at another operator's station — and lets the money and data flow correctly behind the scenes. Two roles do most of the work:

### CPO — Charge Point Operator

Operates the physical stations. Owns the **Locations**, **Sessions**, **CDRs**, and **Tariffs** data. Think EVgo or Electrify America in the US; IONITY or Fastned in Europe.

### eMSP — e-Mobility Service Provider

Has the contract with the driver — the app, the RFID card, the bill. Owns the **Tokens** data. Think ChargePoint's driver app in North America; Shell Recharge or Chargemap in Europe.

Four supporting roles round out the spec: **Hub** (routes traffic between many parties), **NSP** (Navigation Service Provider — surfaces locations in maps), **SCSP** (Smart Charging Service Provider), and **NAP** (National Access Point, a regulatory data aggregator).

## The one idea that prevents most bugs: Sender = data owner

Every OCPI module defines two interfaces — a **Sender** and a **Receiver**. The Sender is the party that *owns and is the source of truth* for that data; the Receiver consumes it. This flips per module:

- For **Locations, Sessions, CDRs, Tariffs** the `CPO is Sender` — the operator owns the stations and what happens at them.
- For **Tokens** the `eMSP is Sender` — the provider owns the driver's credentials.

> **GOTCHA**
>
> Reversing Sender and Receiver is the single most common OCPI integration bug. If you're a CPO, you are the *Sender* of Locations but the *Receiver* of Tokens. Get this backwards and you'll build the wrong half of the interface.

02 · THE MODULE MAP

# All 12 modules at a glance

Who owns the data (Sender), who consumes it (Receiver), and what the module is for. Credentials and Versions are foundational — nothing else works until they do.

| Module | Sender | Receiver | What it carries |
|---|---|---|---|
| **Versions** | both | both | Discovery — supported OCPI versions and each module's endpoint URLs |
| **Credentials** | both | both | The registration handshake and token exchange |
| **Locations** | CPO | eMSP | Charge points — Locations → EVSEs → Connectors, with live status |
| **Sessions** | CPO | eMSP | Charging sessions as they happen, with running energy and cost |
| **CDRs** | CPO | eMSP | Charge Detail Records — the final, billable receipt for a session |
| **Tariffs** | CPO | eMSP | Pricing — energy, time, flat, and parking components |
| **Tokens** | eMSP | CPO | The RFID cards / app tokens the CPO authorizes at the station |
| **Commands** | eMSP | CPO | Remote actions — START, STOP, RESERVE_NOW, UNLOCK_CONNECTOR |
| **ChargingProfiles** | SCSP | CPO | Smart-charging setpoints — shape power over time |
| **HubClientInfo** | Hub | parties | Which parties are currently connected and reachable |

**KEY**

A "party" is identified by `country_code` + `party_id` (e.g. `US` + `EVG`, or `DE` + `ION`). Every object is scoped to its owning party — the same `id` under two different parties are two different objects.

Two parties don't have to implement every module — during the handshake each advertises exactly which modules (and which interface, Sender or Receiver) it supports, so you only build what your role needs.

# Versions + Credentials: the three-token dance

Before any business data flows, two parties negotiate. Call them **Party A** (issues the bootstrap token) and **Party B** (registers). Three tokens are in play — A, B, and C — and each is used by a specific side.

1. **Out-of-band: Party A sends Token A**
   A emails or hands B a one-time *Token A* plus A's Versions URL. This is the only step that doesn't happen over the API.

2. **B discovers A's versions**
   B calls `GET /versions` then `GET /versions/2.2.1` on A, authenticating with Token A, to learn A's supported versions and module endpoints.

3. **B registers with A**
   B calls `POST /credentials` on A (still using Token A). In the body, B includes its own Versions URL and a freshly generated *Token B* — the token A will use to call B.

4. **A responds with Token C**
   A stores Token B, fetches B's versions to discover B's modules, then returns its *credentials* object containing a freshly generated *Token C* — the token B will use to call A.

5. **Token A is retired**
   The moment that `POST /credentials` succeeds, Token A is invalidated. From now on:
   **Party A authenticates to B with Token B; Party B authenticates to A with Token C.**

**GOTCHA**

Token A is single-use, for registration only. There is no "log back in with Token A." To reconnect after a `DELETE /credentials`, the parties must exchange a brand-new Token A out-of-band and run the whole handshake again.

**VERSION NEGOTIATION**

Both sides list every OCPI version they speak. They use the **highest version both support**. If A speaks 2.1.1 + 2.2.1 and B speaks 2.2.1 + 2.3.0, they roam on 2.2.1.

# Auth, addressing & the response envelope

## The Authorization header

Every call carries a token in the header. In 2.2.1 the token is **Base64-encoded** — a change from 2.1.1, where it was sent verbatim.

REQUEST HEADER — OCPI 2.2.1

```
Authorization: Token bXktY3JlZGVudGlhbHMtdG9rZW4=
X-Request-ID: 8f4c2a1e-...        # echoed back, for tracing
X-Correlation-ID: b2d9f0c7-...    # stable across a logical flow
```

> **GOTCHA**
>
> Forgetting to Base64-encode the token (or double-encoding it) yields a 401 that looks like a bad credential. If you migrated from 2.1.1, this is the first thing to check.

## Object addressing

Sender-interface URLs are scoped by owning party: `{module}/{country_code}/{party_id}/{object_id}`. So a specific location owned by US party `EVG` lives at `/locations/US/EVG/LOC1` on the CPO's Sender interface.

## The response envelope

Every OCPI response — success or failure — wraps its payload in the same envelope. The HTTP status tells you the transport worked; the `status_code` tells you whether OCPI is happy.

RESPONSE BODY

```
{
  "data": { /* the object or array */ },
  "status_code": 1000,
  "status_message": "Success",
  "timestamp": "2026-07-08T14:05:22Z"
}
```

**1000** = success · **2xxx** = client error (bad request) · **3xxx** = server error (can't fulfil). A 2.2.1 server can return HTTP 200 with a `status_code` of 2001 — always read both.

05 · WORKED EXAMPLE

# Locations — a full object

A Location contains EVSEs; each EVSE contains Connectors. Nothing here is trimmed "for brevity" — these are the fields a real CPO sends. Note where `status` lives.

GET /LOCATIONS/US/EVG/LOC1 → 200

```
{
  "country_code": "US",
  "party_id": "EVG",
  "id": "LOC1",
  "publish": true,
  "name": "Downtown Garage",
  "address": "200 Main St",
  "city": "Austin",
  "postal_code": "78701",
  "country": "USA",
  "coordinates": { "latitude": "30.26715", "longitude": "-97.74306" },
  "evses": [{
    "uid": "EVG*LOC1*E1",          # stable internal key — use in URLs
    "evse_id": "US*EVG*E1234",        # human/ISO EVSE ID — may change
    "status": "AVAILABLE",          # <-- status lives HERE, on the EVSE
    "connectors": [{
      "id": "1",
      "standard": "IEC_62196_T1_COMBO",  # CCS1 (North America)
      "format": "CABLE",
      "power_type": "DC",
      "max_voltage": 920,
      "max_amperage": 500,
      "max_electric_power": 350000,
      "tariff_ids": ["TARIFF_DC"],
      "last_updated": "2026-07-08T13:55:10Z"
    }],
    "last_updated": "2026-07-08T13:55:10Z"
  }],
  "time_zone": "America/Chicago",
  "last_updated": "2026-07-08T13:55:10Z"
}
```

**GOTCHA — CONNECTORS HAVE NO STATUS**

A Connector has no `status` field. Availability is an **EVSE-level** concept. If your model tracks per-connector status, you'll never find it in OCPI — read the parent EVSE's `status` instead. And address EVSEs by `uid`, not `evse_id`.

06 · WORKED EXAMPLE

# Sessions & Tariffs

A Session is pushed while charging is live (`status: ACTIVE`), then finalized. `end_date_time` and `total_cost` are absent until it completes.

SESSIONS — ACTIVE SESSION (SENDER PUSHES VIA PUT)

```
{
  "country_code": "DE",
  "party_id": "ION",
  "id": "SES-9f31",
  "start_date_time": "2026-07-08T13:40:02Z",
  "kwh": 18.4,
  "cdr_token": {
    "country_code": "DE", "party_id": "SHR",
    "uid": "04A1B2C3", "type": "RFID",
    "contract_id": "DE-SHR-C012345-9"
  },
  "auth_method": "AUTH_REQUEST",
  "location_id": "LOC7",
  "evse_uid": "ION*LOC7*E2",
  "connector_id": "1",
  "currency": "EUR",
  "status": "ACTIVE",
  "last_updated": "2026-07-08T13:55:10Z"
}
```

## Tariffs and the step_size trap

A Tariff's `price_components` each declare a `type`, `price`, `vat`, and `step_size` — the billing increment. The unit of `step_size` depends on the type:

```
{
  "type": "ENERGY", "price": 0.35, "vat": 19.0, "step_size": 1000  # Wh → bill per 1 kWh
}
{
  "type": "TIME",   "price": 0.10, "vat": 19.0, "step_size": 60    # seconds → bill per minute
}
```

> **GOTCHA — MIXED UNITS**
>
> For `ENERGY`, `step_size` is in **Wh**. For `TIME` and `PARKING_TIME`, it's in **seconds**. Treating a TIME step_size as minutes (or an ENERGY step as kWh) silently mis-bills every session.

07 · DATA FLOW

# Push vs Pull — and PUT vs PATCH

The same objects move two ways. Which you use is a design choice per module, not a different data model.

### Pull — the Receiver asks

The Receiver calls `GET` on the Sender's interface, on its own schedule. Simple, but always a little stale, and it hammers the Sender if you poll aggressively.

### Push — the Sender tells

The Sender calls `PUT` / `PATCH` on the Receiver's interface the moment something changes. Fresh and efficient, but the Receiver must run a reliable, always-on endpoint.

## PUT replaces · PATCH merges

- **PUT** sends the *entire* object and replaces it wholesale. Omit a field and you've deleted it.
- **PATCH** sends *only* the changed fields — plus a fresh `last_updated`. Use it for a status flip or a single connector update.

PATCH /LOCATIONS/US/EVG/LOC1/EVG*LOC1*E1 — JUST THE STATUS

```
{ "status": "CHARGING", "last_updated": "2026-07-08T14:01:00Z" }
```

**GOTCHA**

Sending a PATCH-shaped partial body with a PUT wipes every field you left out. Always match the verb to the intent, and always bump `last_updated` so the Receiver can order updates.

**KEY — THERE IS NO PROTOCOL SLA**

OCPI sets no numeric freshness deadline. "Update within a minute" targets come from *regulation* (e.g. the EU's AFIR: dynamic data < 1 min, static < 24 h) or your partner contract — not the spec. Don't assume a mandated latency.

# Top 10 gotchas that reach production

| # | The trap | What's actually true |
|---|---|---|
| 1 | Looking for `connector.status` | Status is an EVSE-level field. Connectors don't have one. |
| 2 | Reusing Token A after the handshake | Token A is single-use for registration. Ongoing calls use Token B / C. |
| 3 | Building the wrong interface half | Sender = data owner. CPO sends Locations; eMSP sends Tokens. |
| 4 | Sending the raw token in the header | 2.2.1 Base64-encodes the Authorization token (2.1.1 didn't). |
| 5 | Treating `step_size` as one unit | ENERGY → Wh; TIME / PARKING_TIME → seconds. |
| 6 | Addressing EVSEs by `evse_id` | Use `uid` — it's the stable key. `evse_id` can change. |
| 7 | Assuming a spec latency SLA | OCPI mandates none. Freshness targets come from regulation/contract. |
| 8 | Using PUT for a partial update | PUT replaces the whole object; PATCH merges changed fields. |
| 9 | Trusting HTTP status alone | Read `status_code` too — HTTP 200 can wrap a 2001 error. |
| 10 | Ignoring `country_code` / `party_id` | They scope every object. Same `id`, different party = different object. |

**THE PATTERN BEHIND THE TRAPS**

OCPI reads cleanly, so teams skim it. Almost every one of these is a place where the obvious assumption is wrong. When in doubt, ask two questions: *who owns this data?* and *where does this field actually live?*

# One-page cheat sheet

## Handshake endpoints

| | |
|---|---|
| GET | /versions |
| GET | /versions/2.2.1 |
| POST | /credentials |
| DELETE | /credentials |

## Verbs

| | |
|---|---|
| GET | Pull an object / list |
| PUT | Replace whole object |
| PATCH | Merge changed fields |
| DELETE | Remove / deregister |

## Status codes

| | |
|---|---|
| 1000 | Success |
| 2000 | Client error (generic) |
| 2001 | Invalid / missing parameters |
| 2002 | Not enough information |
| 3000 | Server error (generic) |
| 3002 | Unsupported version |
| 3003 | No matching endpoints |

## Object hierarchy

```
Location
 └─ EVSE        (has status)
     └─ Connector  (no status)
```

AUTH HEADER · REQUEST TRACING

```
Authorization: Token <base64(credentials_token)>
X-Request-ID · X-Correlation-ID   # echo on every call
```

Party = `country_code` + `party_id`. Sender-URL shape: `/{module}/{country_code}/{party_id}/{id}`.

Learn EV Charging

KEEP GOING

# You've got the map. Now go deeper.

Every module in this guide has a full worked walkthrough — request, response, and the edge cases — on the site (tap to read):

- The Locations module — the data model and real-time push updates
- The Sessions module — live charging sessions, field by field
- OCPI vs OCPP vs ISO 15118 — how the protocols fit together
- Every OCPI guide — the full library as it grows

Free, vendor-neutral guides and tools for the people building EV charging — engineers, PMs, and operators.

learnevcharging.com/blog/ocpi