ENERGY CONNECT

API Getting Started

Energy Connect exposes 122 API endpoints. You need 18 of them for full order control — receiving work, running it to delivery, and posting orders of your own. This guide walks all 18, with examples taken from real calls.

Generated 2026-08-31 (dfb4bf34e)

Always current at energyconnecter.ai/api-getting-started

Machine-readable contract api.energyconnecter.ai/openapi.json

How it works

Energy Connect sits between the company that needs fuel moved and the carrier that moves it. One side posts an **order** — product, quantity, where it's going, when. The other side runs it as a **haul**: accept, dispatch a driver, confirm pickup, deliver.

Both sides use the same API. Which side you're on is a query parameter, not a different endpoint. Carriers read `?role=carrier`; companies posting orders use the default `?role=originator`. Most integrations do one, some do both.

1. Get a key

You generate your own key in the portal — no request to us, no waiting. Sign in as an ADMIN on your company account and go to **Settings → Integrations → API Key**.

Scopes are chosen at creation, so decide what your integration does before you generate one. Adding a scope later means generating a new key.

For a carrier integration that *drives* work rather than just watching it:

`orders:write`	create and update orders (implies orders:read)
`haul_execution:write`	accept, dispatch, confirm pickup, deliver
`webhooks:write`	subscribe to events (implies webhooks:read)
`invoices:read`	read invoices
`locations:read`	delivery locations
`connections:read`	who you are connected to

> **This one trips people up.** `orders:write` does *not* let you accept a load — that is `haul_execution:write`. A key scoped only for reading and invoicing can see work it cannot act on, which looks like a permissions bug and is not one.

Keys are shown **once**, at creation, and stored hashed — they cannot be recovered. `platform:usage_read` is granted automatically on every key.

Environments

ENVIRONMENT	BASE URL	KEY PREFIX
Production	`https://api.energyconnecter.ai/v1/public`	`ec_live_`
Staging	`https://staging-api.energyconnecter.ai/v1/public`	`ec_test_`

Staging holds synthetic data only — create, cancel and delete freely.

2. Your first call

Every key can call this one, so it is the fastest way to prove your credentials work before you write anything else.

```
curl -H "X-API-Key: ec_live_your_key_here" \
  https://api.energyconnecter.ai/v1/public/usage/me
```

```
{
  "success": true,
  "data": {
    "keyName": "Dispatch integration",
    "mode": "live",
    "burst": { "limit": 600,   "remaining": 599,   "windowSec": 60 },
    "quota": { "limit": 50000, "remaining": 49999, "windowDays": 30 }
  }
}
```

`Authorization: Bearer ec_live_…` works identically if that suits your HTTP client better.

3. Conventions to handle first

RESPONSE ENVELOPE

```
{ "success": true,  "data": { … } }
{ "success": false, "error": {
    "code": "…", "message": "…" } }
```

LIST PAGINATION

```
"meta": { "total": 27, "page": 1,
  "pageSize": 20, "totalPages": 3,
  "hasMore": true }
```

`Idempotency-Key` **is required on every mutating request** — POST, PUT, PATCH and DELETE, not just some of them. Omit it and you get `IDEMPOTENCY_KEY_REQUIRED`.

Replay the same key and you get the cached response with `Idempotency-Replayed: true`, and **side effects do not re-fire** — no duplicate dispatch SMS, no duplicate emails. Retrying is safe.

Errors tell you what to fix

Validation failures name the exact field path:

```
{ "code": "VALIDATION_ERROR", "details": [
    { "field": "lines.0.product",  "message": "expected string, received undefined" },
    { "field": "lines.0.quantity", "message": "expected number, received undefined" } ]
}
```

Permission failures name the exact scope you are missing:

```
{ "code": "INSUFFICIENT_SCOPE",
  "message": "Missing required scope: company_catalog:read",
  "details": { "requiredScope": "company_catalog:read" } }
```

Rate limits are 600 requests/minute burst and 50,000 billable per month, with `RateLimit-*` headers on every response.

4. Reading the address

An order's delivery address is stored one of two ways, depending on how it was created. **They are mutually exclusive**, and which one you get is decided by whoever posted the order — not by you.

ORDER CREATED WITH	LOCATION	DESTINATIONSTREET
a saved location	populated object	`null`
a typed address	`null`	populated

Read `fullDestinationAddress`. It is assembled from whichever source the order actually has, and it is the only address field that works for both shapes.

An integration that reads only `destinationStreet` works perfectly until a counterparty starts using saved locations, and then silently returns nothing. The same applies to the customer name: `destinationName` on one shape, `location.name` on the other.

```
"fullDestinationAddress": "1101 River Ave, Houston, TX 77002"
```

5. Running a haul

What is waiting for you

```
curl -H "X-API-Key: $KEY" \
  "https://api.energyconnecter.ai/v1/public/orders?role=carrier"
```

Only accepted loads appear here. An order reaches your dispatch system once your company has committed to it — offers you have not accepted are not delivered. To see pending offers, ask for them explicitly with `?status=ASSIGNED`; their delivery address and customer name stay withheld until acceptance.

> **`ASSIGNED` means "pending your acceptance", not "assigned to you".** Our portal labels this state *Pending Acceptance*. A load sitting in `ASSIGNED` has been offered — the poster may well describe it as "sent to you" — but nothing moves until someone accepts it.

One haul in detail

```
curl -H "X-API-Key: $KEY" \
  "https://api.energyconnecter.ai/v1/public/orders/{id}?role=carrier"
```

> **`?role=carrier` is required here too.** Without it the read defaults to orders *your* company posted, and a haul you were assigned returns **404 — "Order not found"**. The record exists; you asked as the wrong party. If a detail read 404s on an id you just listed, this is why.

The sequence

In order. Each call requires an `Idempotency-Key`.

```
POST /orders/{id}/accept            {}  or  {"eta": "…"}    ASSIGNED → CONFIRMED
POST /orders/{id}/dispatch          {"selfDispatch": true}
POST /orders/{id}/confirm-pickup    {bols[], lines[]}        → IN_TRANSIT
POST /orders/{id}/delivery          {gallonsDelivered, …}
POST /orders/{id}/notify-delivered  {}                       → PENDING_VERIFICATION
```

Some posters require an ETA to accept their loads — the 400 tells you so; resend with `eta` (ISO datetime). Confirming pickup moves the load to `IN_TRANSIT` on its own: a loaded truck is delivering. `POST /orders/{id}/start-transit` still exists and is harmless — on a load already in transit it returns 200 with `alreadyInTransit: true` — but you do not need to call it.

> **The lifecycle does not end at `DELIVERED`.** Submitting delivery moves the load to `PENDING_VERIFICATION` — the poster still verifies it. If you poll waiting for `DELIVERED`, you will wait forever.

Calling a step out of order returns a 409 that says exactly what is wrong:

```
"Order is CONFIRMED/DISPATCHED — start-transit only valid
 after pickup is confirmed (load complete)."
```

Confirming pickup carries the BOLs

A bill of lading is issued **per pickup stop**, so one load can carry several. Send them as an array:

```
{
  "bols":  [{ "number": "BOL-0001", "cardInTime": "09:12", "cardOutTime": "09:58" }],
  "lines": [{ "orderLineIndex": 0, "netGallons": 7795, "grossGallons": 7800 }],
  "sealNumbers": ["S-1101"]
}
```

Loaded and delivered volumes are **driver-reported actuals**. `net > gross` is legal — cold-fuel temperature correction — and nothing is cross-checked against the ordered quantity.

Attaching a BOL or POD

1. `POST /uploads` → returns `{ id, uploadUrl, fileUrl, expiresIn }`
2. `PUT <uploadUrl>` → the file bytes, straight to storage
3. `POST /orders/{id}/documents` → `{ docType, fileUploadId }`

> Two things worth knowing: step 1 requires `fileSize` in the body, and it returns the identifier as `id` — but step 3 expects it as `fileUploadId`. Same value, different names.

Document URLs are short-lived and re-minted on every read. Persist `downloadUrl`, never `fileUrl`.

After delivery: the invoice

Once the poster verifies the delivery, billing happens on the same API. `GET /invoices` lists yours — each carries `orderId`, so reconciling an invoice back to the haul it bills is one field, not a lookup.

6. Posting orders in

`POST /orders`	201, status DRAFT
`PUT /orders/{id}`	amend while DRAFT
`POST /orders/{id}/publish`	DRAFT → POSTED
`POST /orders/{id}/cancel`	→ CANCELLED
`GET /orders`	your own posted orders

> **Orders are created as `DRAFT`.** `publish` is a separate, required call — a POST on its own creates something no carrier can see, and still returns 201, so nothing signals the omission.

The minimum body is not a useful order

The API accepts far less than a dispatchable order needs. This is the *entire* required body:

```
{ "lines": [{ "product": "Diesel #2", "quantity": 7500 }] }
```

That succeeds with no destination, no date and no customer — producing an order with `fullDestinationAddress: null` that nobody can deliver. Send a complete order instead:

```
{
  "destinationStreet": "1101 River Ave",
  "destinationCity":   "Houston",
  "destinationState":  "TX",
  "destinationZip":    "77002",
  "requestedDate":     "2026-08-12T14:00:00Z",
  "lines": [{ "product": "Diesel #2", "quantity": 7500, "unitCode": "gal" }]
}
```

…or reference a saved location instead of typing the address:

```
{ "locationId": "e5f6a7b8-…", "lines": [{ "product": "Diesel #2", "quantity": 7500 }] }
```

Remember section 4 — that choice determines which address fields your own reads return afterwards.

Two reads support this flow: `GET /locations` lists your saved delivery locations — it is where the `locationId` above comes from — and `GET /connections` lists the companies you are connected to, which is who your published orders can reach.

Units. `unitCode` accepts any code from `GET /product-taxonomy/units`. `gallons` and `unit` are permanent aliases of `quantity` and `unitCode` — both supported forever; prefer the newer names.

Your own orders are never masked. Reading orders you posted, you always see everything, at every status.

7. Staying in sync

The list endpoint returns your whole history. Polling it to notice one new load means paging through everything you have ever run, on an interval, forever. Subscribe instead.

Subscribe

```
curl -X POST -H "X-API-Key: $KEY" -H "Content-Type: application/json" \
  -H "Idempotency-Key: $(uuidgen)" \
  -d '{"url": "https://your-system.example.com/hooks/ec",
       "events": ["order.updated"]}' \
  https://api.energyconnecter.ai/v1/public/webhooks
```

The signing secret is returned **once**, on create. Your URL must be `https` and publicly routable.

```
{
  "id": "9b2fa1c4-…",
  "type": "order.updated",
  "createdAt": "2026-08-07T14:02:11Z",
  "data": { "orderId": "a1b2c3d4-…", "status": "CONFIRMED" }
}
```

Payloads are deliberately thin — ids and status only. The value is the *timing*, not the contents. Fetch the order through the API when an event arrives, so you receive it with the correct permissions applied rather than a snapshot that may already be stale.

`order.updated` **fires on *every* status change, including offers you have not accepted.** If you only want work you have committed to, subscribe to `order.accepted` instead — it fires once, on `ASSIGNED → CONFIRMED`, so every fetch it triggers is post-acceptance. Acting on `order.updated` without checking the status afterwards means pulling a load that is still pending acceptance, finding its address and customer withheld by design, and concluding the data is broken.

The full catalogue is `order.updated`, `order.accepted`, `bid.created`, `bid.accepted`, `bid.declined`, `bid.countered`, `bid.withdrawn`, `invoice.issued`, `invoice.paid`, `delivery_ticket.signed` and `bol.submitted`. Subscribe to as many as you need in a single subscription.

Verifying the signature

Every delivery carries `X-EC-Event-Id`, `X-EC-Delivery-Id`, `X-EC-Timestamp` and:

```
X-EC-Signature: v1=hex(HMAC_SHA256(secret, "{timestamp}.{rawBody}"))
```

Compute it over the **raw** body — not a re-serialised object, or it will never match — and compare in constant time. Reject anything with a timestamp more than five minutes old to block replays.

Delivery guarantees

- **At-least-once**, with a stable event id. Deduplicate on `id` — you will occasionally see the same event twice.
- A `2xx` within 10 seconds counts as delivered. Anything else retries at 1m, 5m, 30m, 2h and 12h, then dead-letters.
- Inspect failures with `GET /webhooks/{id}/deliveries`.
- Catch up after an outage with `POST /webhooks/{id}/replay` — up to 7 days back, 1000 events, same ids as the originals.

Still want to poll?

Reasonable if your network cannot expose an inbound endpoint. Use `GET /orders?role=carrier` — carrier reads are sorted most-recently-updated first, so page 1 is where change appears.

8. Going live

- **Scope narrowly.** Separate keys for separate jobs; a key that only reads should not hold write scopes.
- **IP allowlisting** is available per key — unlisted addresses get a `403`.
- **Shadow mode** lets you profile your traffic before going live: usage is recorded, rate limits are not enforced.
- **Rotate and revoke yourself**, in the same place you generated the key. Rotating issues a replacement and leaves the old key valid for **24 hours** so you can cut over without downtime; revoking kills a key immediately and cannot be undone. If a key is ever exposed, revoke first, then generate a replacement.
- **Changes are additive.** Renamed fields keep their old names permanently as documented aliases, so existing payloads keep working.

Browse all 122 endpoints in the full API reference →