Connect your POS to DrippleX

A step-by-step guide for merchants and point-of-sale vendors.

Base URL `https://api.dripplex.com/api/v1` · Interactive reference `api.dripplex.com/docs/pos` · September 2026

What this integration does

Once connected, your point of sale can:

- push your product catalogue into DrippleX, so customers see what you actually sell;
- push stock levels, so nobody orders what you have run out of;
- read DrippleX orders as they arrive, without anyone retyping them;
- tell DrippleX when an order is being prepared and when it is ready.

> **What it does not do, so you can plan around it.** DrippleX never calls your server — there are no webhooks, and your POS learns about orders by polling. Your POS cannot create a DrippleX order, cannot take a DrippleX payment, and does not drive settlement. Orders start with the customer in the DrippleX app and are paid there before they reach you.

Before you start

- An approved DrippleX merchant account.
- Somewhere server-side to keep an API key. Not a browser, not a mobile app.
- A stable identifier for each product in your own system — your SKU. DrippleX matches on it, so it must not change between syncs.

1 Create your integration

Sign in to your DrippleX merchant account and create an integration. You do not need anyone at DrippleX to provision it for you.

DrippleX returns two things:

`integrationId`	Identifies this connection. Safe to log.
`apiKey`	Authenticates it. Secret.

> **The key is shown once.** DrippleX stores only a hash of it, so we cannot show it to you again — only replace it. Copy it into your secret store now. If it leaks, rotate it; the old key stops working immediately.

2 Authenticate every request

Two headers. There is no bearer token and no OAuth flow.

```
curl https://api.dripplex.com/api/v1/integrations/orders/list \
  -H "x-integration-id: YOUR_INTEGRATION_ID" \
  -H "x-integration-key: YOUR_API_KEY"
```

Your key carries *scopes*, and each endpoint requires one:

Scope	Lets you
`catalog:write`	Push your catalogue
`inventory:write`	Push stock levels
`orders:read`	List and read orders
`orders:write`	Update an order's status

> **Your key is for pushing and for orders — not for reading back.** Your credential is also issued `catalog:read` and `inventory:read`, which no endpoint reads today. Reviewing what DrippleX imported — sync history, imported products, stock levels, category mappings and conflicts — is done by a person signed in to DrippleX, not by your key. Sending the key to one of those returns `401`. We would rather tell you here than have you find it mid-build.

`401` means the key did not authenticate. `403` means it did, but lacks the scope. They are different answers to different questions.

> **No API on the till?** The merchant can still get their catalogue in. A CSV export imported from the DrippleX merchant portal runs through this same pipeline — same SKU identity, same category mappings, same draft-then-publish, same idempotency. It is documented at `https://dripplex.com/developers` under “If the till has no API”. Steps 3 and 4 below apply either way.

---

3 Map your categories

Do this once, before your first catalogue push. It takes a few minutes and it decides whether your products are findable.

Each item you send carries your own category name in `categoryName`. DrippleX will not invent a category from it. **Until that name is mapped, the products carrying it import uncategorised** and each one raises a `CATEGORY_UNMAPPED` conflict. The sync still succeeds and nothing is lost — the products simply will not sit anywhere a customer browses.

First, list the DrippleX categories and their ids. This endpoint is public:

```
curl https://api.dripplex.com/api/v1/categories
```

Then point each of your category names at one of them. This is done signed in as the merchant, from the DrippleX merchant portal or with a merchant session token:

```
PUT /integrations/catalogue/mappings/{integrationId}

{
  "externalCategoryName": "Pastries",
  "categoryId": "305c2d0d-00e5-4f43-935d-c5852c885300"
}
```

Several of your categories may point at the same DrippleX one. A bakery mapping *Cakes*, *Pastries*, *Snacks* and *Cookies* all to *Restaurants & Food* is normal. A pharmacy would instead map *OTC Medicines* to *Pharmacy & Health*, *Baby Care* to *Baby, Kids & Toys* and *Toiletries* to *Beauty & Personal Care*. The mapping describes your shop; there is no fixed list to conform to.

> **The name is matched exactly as your POS spells it** — case and spacing included. `Pastries` and `pastries` are two different mappings. Normalising them here would quietly stop matching what your POS actually sends, which is worse than an obvious mismatch you can see and fix.

The call is idempotent on the pair *(integration, category name)*, so re-pointing a category later is the same request as mapping it the first time and a retry is never an error. Add a new category at the till? Map it before its products sync — or map it afterwards and re-send them. The conflicts clear either way.

---

4 Push your catalogue

`POST /integrations/catalogue/sync` · scope `catalog:write` · up to **500 items** per batch.

```
{
  "idempotencyKey": "catalogue-2026-09-22-0900",
  "items": [
    {
      "externalSku": "JOLLOF-LG",
      "name": "Jollof Rice (Large)",
      "description": "Party jollof with chicken",
      "price": 4500.00,
      "currency": "NGN",
      "quantity": 40,
      "categoryName": "Rice dishes",
      "active": true
    }
  ]
}
```

> **The idempotency key is doing real work.** If your POS retries after a timeout, replaying the same key returns the original job untouched instead of importing everything twice. Derive it from something stable — a batch id or a timestamp — not a random value generated at send time.

The response tells you the outcome directly — there is nothing to poll:

```
{
  "jobId": "244a5a3a-803a-4f26-96b1-b9cae120913d",
  "jobStatus": "COMPLETED",
```

```
  "productCount": 18,
  "failedCount": 0,
  "replayed": false
}
```

`replayed: true` means DrippleX recognised the idempotency key and returned the original job rather than importing anything a second time. The fuller history, at `GET /integrations/catalogue/jobs/{integrationId}`, is read by a signed-in merchant in the portal — not by your key.

---

5 Push stock levels

`PUT /integrations/inventory/sync` · scope `inventory:write` · up to **500 items** per batch.

```
{
  "items": [
    { "externalSku": "JOLLOF-LG", "quantity": 12 },
    { "externalSku": "SUYA-STD",  "quantity": 0 }
  ]
}
```

Send only what changed. A quantity of `0` takes an item out of stock — it does not delete it.

---

6 Receive orders by polling

`GET /integrations/orders/list` · scope `orders:read`

DrippleX does not call your server. Your POS asks. Every 30 to 60 seconds suits most kitchens and counters; poll faster and you will meet `429`.

Read one order in full with `GET /integrations/orders/detail/{orderNumber}`.

The money on an order

An order carries `subtotal`, `discount`, `tax` and `total`. It does not break out DrippleX's delivery fee or DX fee: those are our economics and are not needed to fulfil an order. The difference is derivable in aggregate, and we would rather tell you than have you work it out:

```
total − (subtotal − discount + tax) = DrippleX fees on this order
```

That residual is the delivery fee *plus* the DX fee, each net of any DrippleX promotion. It is not the delivery fee on its own.

---

7 Update the order status

`PUT /integrations/orders/status/{orderNumber}` · scope `orders:write`

```
{
  "externalOrderId": "POS-88412",
```

```
  "status": "PREPARING",
  "sourceTimestamp": "2026-09-22T09:14:00Z"
}
```

Your POS can drive exactly two statuses:

Status	Means	Allowed from
`PREPARING`	You accepted it and started	A confirmed order
`READY`	It is ready for collection or a rider	An order already preparing

> Everything else — delivery, completion, cancellation — stays with DrippleX. Cancelling refunds the customer, so it is deliberately not something a POS can request.

When something goes wrong

Code	What it means	What to do
`400`	A field in your body is invalid	Read the message; it names the field
`401`	The key did not authenticate	Check both headers; the key may be rotated or revoked
`403`	Authenticated, but missing the scope	Issue a key with the scope you need
`404`	No such order or job, or not yours	Check the identifier
`429`	Too many requests	Slow your polling down

Every way a key can fail to authenticate returns the same `401`. That is deliberate: a more specific error would let someone probe for valid integration ids.

Before you go live

- The API key is server-side, in a secret store — not in source control.
- Every category your POS sends is mapped, and `GET /integrations/conflicts/{integrationId}` comes back empty.
- Your SKUs are stable and will not change between syncs.
- Catalogue pushes use an idempotency key derived from something repeatable.
- Order polling runs on a timer, handles `429`, and backs off.
- Your POS marks `PREPARING` on accept and `READY` when it is ready.
- You know how to rotate the key, and somebody other than one person can do it.
