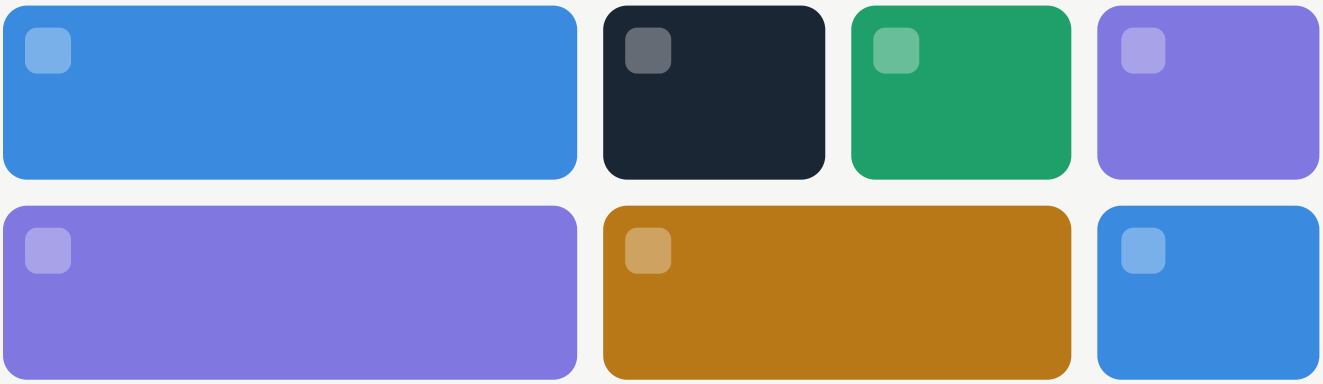
BPMSquare

CUSTOMER EDITION

Recipes

How to connect other systems to BPMSquare: the REST API, the change feed and webhooks, connectors, the integration middleware and Data Workbench.

New way of customer relationship



Recipes

CHAPTER 7

Recipes

> Screen names and button words may be refined as the product improves. If a name here is slightly different on your screen, look for the same idea nearby.

About this chapter

This chapter puts the earlier chapters to work. It has four recipes. Each one starts from nothing and ends with a check that proves it worked.

1. Keep an ERP in step with your accounts, using the change feed.
2. Receive a webhook and process it safely.
3. Create a quotation from your own system and read its status back.
4. Load 5,000 products with Data Workbench.

The code is Node.js 18 or later. Only the receiver in recipe 2 needs a package (`express`). Every example reads its key or secret from a small text file next to the script, such as `bpm.key`. Keep those files out of your source repository. The workspace address `bpm.example.com`, the customer Example Trading LLC and the ids are made up. Use your own.

Before you start any recipe, read Chapter 1, sections 3 and 6.

Contents

1. Keep an ERP in step with accounts
2. Verify and process a webhook safely
3. Create a quotation and read its status back
4. Load 5,000 products with Data Workbench
5. Common questions

---

1. Keep an ERP in step with accounts

Goal. Every account in BPMSquare exists in your ERP, and every change or deletion reaches it within minutes. BPMSquare is the source of truth. Your ERP only receives.

You need.

- A scoped key, **Read** only, with the single object **Accounts**. Name it "ERP account sync".
- A way to create or update an account in your ERP by BPMSquare's account `id`. Store that id in a field of your ERP record.

- A job scheduler that can run a script every five minutes.

The plan.

1. Note the time. Read every account once with the list endpoint and the detail endpoint, and write each to the ERP.
2. Then read the change feed for accounts, from the beginning. Ignore events older than the time you noted. For each other event, read the account again and write it to the ERP, or remove it if it is gone.
3. Save the cursor after each page. From then on, each run starts from the cursor.

Why read the account again, rather than use the `changes` in the event? Because the event may be redacted (phone and email show as `[redacted]`), and because reading the record again makes every step safe to repeat.

The script

Save as `sync-accounts.mjs`. Put the key on one line in `bpm.key`.

```
import { readFile, writeFile } from "node:fs/promises";

const BASE = "https://bpm.example.com/api/v1";
const KEY = (await readFile("bpm.key", "utf8")).trim();
const STATE_FILE = "accounts.state.json";

async function api(path) {
  const res = await fetch(`${BASE}${path}`, { headers: { Authorization: `Bearer ${KEY}` } });
  if (res.status === 404) return null;
  if (!res.ok) throw new Error(`${res.status} ${path} ${await res.text()}`);
  return res.json();
}

// Replace these two with calls to your ERP. Both must be safe to repeat.
async function upsertInErp(account) {
  console.log("upsert", account.ref, account.name, account.id);
}
async function removeFromErp(id) {
  console.log("remove", id);
}

async function readState() {
  try { return JSON.parse(await readFile(STATE_FILE, "utf8")); } catch { return {}; }
}
const saveState = (s) => writeFile(STATE_FILE, JSON.stringify(s));

async function loadAll() {
  for (let page = 1; ; page++) {
    const list = await api(`/accounts?limit=200&page=${page}&sort=created_at&select=id`);
    for (const row of list.data) {
      const one = await api(`/accounts/${row.id}`);
      if (one) await upsertInErp(one.data);
    }
    if (!list.meta.has_more) break;
  }
}

async function readFeed(state) {
  let cursor = state.cursor ?? null;
  for (;;) {
    const q = new URLSearchParams({ object_type: "accounts", limit: "500" });
    if (cursor) q.set("since", cursor);
    const out = await api(`/changes?${q}`);

    // The last action for each account in this page wins.
    const last = new Map();
    for (const e of out.data) {
      if (new Date(e.at) < new Date(state.floor)) continue; // already covered by the first load
      last.set(e.object_id, e.action);
    }
    for (const [id, action] of last) {
      if (action === "delete") { await removeFromErp(id); continue; }
      const one = await api(`/accounts/${id}`);
      if (one) await upsertInErp(one.data);
      else await removeFromErp(id);
    }
```

```
    cursor = out.cursor;
    await saveState({ ...state, cursor }); // only after the page is done
    state.cursor = cursor;
    if (!out.meta.has_more) break;
  }
}

const state = await readState();
if (!state.loaded) {
  state.floor = new Date().toISOString();
  await loadAll();
  state.loaded = true;
  await saveState(state);
}
await readFeed(state);
```

Run it once by hand: `node sync-accounts.mjs`. The first run loads everything. Run it every five minutes after that.

How it behaves

- If it stops half-way through the first load, the next run starts the load again with a later `floor`. Nothing is lost, and `upsertInErp` is safe to repeat.
- If it stops half-way through a page, the cursor is not saved. The next run reads the same page again.
- An account that was created and deleted between two runs shows as a `delete`. The script removes it from the ERP, which is harmless if it was never there.
- Contact people are not in the `accounts` feed. They come inside `GET /accounts/:id`, under `contacts`. Because the script reads the full account each time, a change to the account is enough to refresh them, but a change to only a contact is not logged under `accounts`. If you need contacts to be fresh, also read the full account on a daily schedule.
- Only fields your key can see arrive. Chapter 1, section 7 says which are personal.

Check it worked.

1. In BPMSquare, rename the account "Example Trading LLC" to "Example Trading LLC (Dubai)".
2. Wait for the next run. Your log shows `upsert ACC-0007 Example Trading LLC (Dubai) …`.
3. Compare totals: `curl -s -H "Authorization: Bearer $KEY" "$BASE/accounts?count=only"` returns `meta.total`. It must equal the number of accounts in your ERP that carry a BPMSquare id.

---

2. Verify and process a webhook safely

Goal. BPMSquare tells your system when quotations change. Your receiver accepts only genuine calls, answers fast, does each piece of work once, and survives a repeat.

You need.

- A public `https` address that reaches your receiver, such as `https://hooks.example.com/hooks/bpmsquare`.
- A workspace administrator to make the webhook.
- A scoped key, **Read** only, for **Quotations**, to read the current state of a quotation.

Set up the webhook.

1. **Settings > Integrations > Webhooks > New webhook** (or **Webhooks & push** if that is what your workspace has).
2. Name it "Quotation changes". Kind **Event**. Under **Changes sent**, tick **Quotes** only.
3. Address: `https://hooks.example.com/hooks/bpmsquare`.
4. Create it. On **Security**, copy the signing secret into a file named `whsec.txt` next to the receiver.

The receiver. Save as `receiver.mjs`. It needs the package `express` (`npm install express`).

```
import crypto from "node:crypto";
import { readFileSync, appendFileSync, existsSync } from "node:fs";
import express from "express";

const SECRET = readFileSync("whsec.txt", "utf8").trim();
const BASE = "https://bpm.example.com/api/v1";
const KEY = readFileSync("bpm.key", "utf8").trim();

// Events already handled. Append-only file so a restart does not forget them.
const SEEN_FILE = "seen-events.log";
const seen = new Set(existsSync(SEEN_FILE) ? readFileSync(SEEN_FILE, "utf8").split("\n").filter(Boolean) :
[]);

const queue = [];
let working = false;

function verify(rawBody, given) {
  const expected = crypto.createHmac("sha256", SECRET).update(rawBody).digest("hex");
  return given.length === expected.length && crypto.timingSafeEqual(Buffer.from(given),
Buffer.from(expected));
}

async function handle(event) {
  if (event.object_type !== "quotes" || event.action === "delete") return;
  // Read the current record. The event is a prompt, not the truth.
  const res = await fetch(`${BASE}/quotations/${event.object_id}`, { headers: { Authorization: `Bearer
${KEY}` } });
  if (!res.ok) throw new Error(`read ${event.object_id}: ${res.status}`);
  const q = (await res.json()).data;
  console.log("quotation", q.ref, "status", q.status, "outcome", q.outcome, "total", q.total);
  // Update your own system here.
}

async function drain() {
  if (working) return;
  working = true;
  while (queue.length) {
    const event = queue.shift();
    try {
      await handle(event);
      appendFileSync(SEEN_FILE, event.id + "\n");
      seen.add(event.id);
    } catch (err) {
      console.error("failed", event.id, err.message);
      // Not marked as seen: it will be handled when BPMSquare sends it again.
    }
  }
  working = false;
}

const app = express();
app.post("/hooks/bpmsquare", express.raw({ type: "application/json", limit: "2mb" }), (req, res) => {
  if (!verify(req.body, req.get("X-BPMSquare-Signature") || "")) return res.status(401).end();

  const batch = JSON.parse(req.body.toString("utf8"));
  for (const event of batch.events || []) {
    if (event.object_type === "ping" || seen.has(event.id)) continue; // a test, or a repeat
```

```
    if (!queue.some((q) => q.id === event.id)) queue.push(event);
  }
  res.sendStatus(200); // acknowledge now; the work happens after
  drain();
});

app.listen(3000, () => console.log("listening on 3000"));
```

Why it is built this way.

- `express.raw` keeps the exact bytes. The signature is checked over them.
- It answers `401` for a bad signature and does nothing else.
- It answers `200` before the work is done, so a slow system never makes BPMSquare time out (10 seconds).
- A piece of work that fails is not marked as seen, but BPMSquare will not send it again, because the receiver already answered `200`. If you cannot afford to lose an event, make `handle` write the event to a durable queue (a database table) before you answer. Answer `5xx` if that write fails. BPMSquare then sends the batch again.
- Each event is handled once because the event `id` is remembered.

Test it without BPMSquare. Save as `send-test.mjs`. It signs a batch with the same secret and sends it three ways.

```
import crypto from "node:crypto";
import { readFileSync } from "node:fs";

const secret = readFileSync("whsec.txt", "utf8").trim();
const url = "http://localhost:3000/hooks/bpmsquare";

const body = JSON.stringify({
  delivery_id: "test-1",
  sent_at: new Date().toISOString(),
  events: [{
    id: "test-event-1",
    object_type: "quotes",
    object_id: "b7d1e0c2-5a3f-4e18-8c92-6f4a1d9e2b03",
    object_label: "QT-2026-0042",
    action: "update",
    changes: [{ field: "status", from: "draft", to: "sent" }],
    actor: "sara.admin@example.com",
    at: new Date().toISOString(),
  }],
});
const sign = (b) => crypto.createHmac("sha256", secret).update(b).digest("hex");

async function send(label, b, signature) {
  const res = await fetch(url, { method: "POST", headers: { "Content-Type": "application/json", "X-
BPMSquare-Signature": signature }, body: b });
  console.log(label, res.status);
}

await send("valid     ", body, sign(body));
await send("repeat    ", body, sign(body));
await send("tampered  ", body.replace("sent", "send"), sign(body));
```

Put the id of a real quotation of your workspace in `object_id` first. Otherwise the read of the quotation fails and the receiver logs `failed` instead.

Expected output: `valid 200`, `repeat 200`, `tampered 401`. The receiver logs the quotation line once, because the repeat is skipped, and logs nothing for the tampered request.

Check it worked on the real thing.

1. On the webhook's **Test** tab, press the test. The receiver logs nothing for a test message and answers `200`. The webhook's row shows the last call as successful.
2. Change a quotation's status in BPMSquare. At the next delivery your receiver logs the quotation. Ask your BPMSquare contact how often deliveries run.
3. Send the `tampered` request again. It must be `401`.
4. Restart the receiver and send the `valid` request again. It must be skipped, because `seen-events.log` remembers it.

---

3. Create a quotation and read its status back

Goal. Your order system creates a quotation in BPMSquare for a customer, follows it until the customer decides, and shows its invoiced and paid amounts.

You need.

- A scoped key with **Read** and **Write**, for **Quotations** and **Accounts**. Name it "Order system".
- The customer already exists in BPMSquare. If not, create it on screen, or load it with Data Workbench (Chapter 6).

The plan.

1. Find the account's `id` by name.
2. `POST /quotations` with your own order number in `ref_no`. Store the `id` that comes back.
3. Read the quotation by `id` for its status.
4. Read the quote-to-cash figures from the list.

The script

Save as `quote.mjs`.

```
import { readFile } from "node:fs/promises";

const BASE = "https://bpm.example.com/api/v1";
const KEY = (await readFile("bpm.key", "utf8")).trim();

async function call(method, path, body) {
  const res = await fetch(`${BASE}${path}`, {
    method,
    headers: { Authorization: `Bearer ${KEY}`, "Content-Type": "application/json" },
    body: body ? JSON.stringify(body) : undefined,
  });
  const json = await res.json().catch(() => ({}));
  if (!res.ok) {
    const detail = json.details ? " " + JSON.stringify(json.details) : "";
    throw new Error(`${method} ${path} -> ${res.status} ${json.error ?? ""}${detail}`);
  }
  return json;
}

// 1. Find the account by name.
const q = new URLSearchParams({ filter: "name:eq:Example Trading LLC", select: "id,name", limit: "2" });
const found = (await call("GET", `/accounts?${q}`)).data;
if (found.length !== 1) throw new Error(`expected one account, found ${found.length}`);
const accountId = found[0].id;

// 2. Create the quotation. Your own order number goes in ref_no.
const created = (await call("POST", "/quotations", {
  account_id: accountId,
  name: "Supply of 22 kW motors and installation",
  ref_no: "ORD-88123",
  quote_date: "2026-10-26",
  valid_until: "2026-11-25",
  gst_rate: 5,
  terms: "Payment 30 days from invoice.",
  lines: [
    { sl_no: "1", description: "Supply of 3-phase induction motor 22 kW", uom: "Nos", qty: 2, rate: 18500,
category: "material" },
    { sl_no: "2", description: "Installation and commissioning", uom: "Lot", qty: 1, rate: 4500,
discount_pct: 10, category: "labour" },
  ],
})).data;
console.log("created", created.ref, created.id, "total", created.total, "status", created.status);

// 3. Read it back by id.
const one = (await call("GET", `/quotations/${created.id}`)).data;
console.log("status", one.status, "outcome", one.outcome, "valid until", one.valid_until);

// 4. Quote to cash, for this account.
const cash = (await call("GET", `/quotations?filter=${encodeURIComponent("account.id:eq:" +
accountId)}&select=ref,status,outcome,total,invoice_count,invoiced_total,paid_total,balance_due`)).data;
console.table(cash);
```

Run `node quote.mjs`. The first line shows the new reference (for example `QT-2026-0043`) and its id, `total 41050`, and status `draft`.

Following it afterwards

- **By id.** Keep the `id` with your order number. Read `GET /quotations/:id`. It carries `status`, `outcome` (`open`, `won`, `lost`, `dropped`), `total`, `revision` and `superseded_by`.
- **By the change feed.** Read `GET /changes?object_type=quotes` with a stored cursor (Chapter 3). A quotation moved to won or lost shows as an update to `outcome`. Match on `object_id`.
- **Revisions.** If someone revises the quotation, the older version gets `superseded_by` set and becomes read-only. A `PATCH` on it answers `409`. Follow `superseded_by` to the newer version.
- **Changing it.** `PATCH /quotations/:id` with only the fields to change. If you send `lines`, send the complete set. They replace the old lines. If the call fails, read the quotation back before you repeat it.
- **Money.** `total` is before tax. `invoiced_total`, `paid_total` and `balance_due` come from the quotation's own invoices that are not cancelled.

A `POST` is not repeatable

If the call times out, you do not know if the quotation was created. Do not simply send it again, or you may create two.

1. List the account's quotations created since you started: `GET /quotations?filter=account.id:eq:<id>;created_at:gte:<time you started>&select=id,ref`.
2. Open each with `GET /quotations/:id` and look for your `ref_no`.
3. Create it again only if none carries it.

Check it worked.

1. The script prints a reference and an id.
2. In BPMSquare, open the quotation. The title, the two lines and the total of 41,050 are there.
3. In **Change history**, the quotation is recorded as created by `api:v1`.
4. Mark it sent on screen. `GET /quotations/:id` now shows `status: "sent"`.

---

4. Load 5,000 products with Data Workbench

Goal. Put a 5,000-item catalogue into **Products**, once, correctly, with prices and quantity breaks.

You need.

- The Products and Data Workbench modules, and a business role that includes Data Workbench.
- Your product category tree set up under **Settings** (categories and sub-categories). The Category and Sub-category columns take the codes from that tree.
- Suppliers loaded first, if you want to name a preferred supplier in the file. The supplier must exist already, spelled the same.
- A read-only API key with the **Products** object, to check the result.

Steps.

1. **Get the template.** Open **Admin > Data Workbench > Import > Products**. Press **Download Products template**. It carries every column of your workspace, your custom fields included, with example rows.
2. **Prepare the data in a spreadsheet.** The main columns are:

Column	Notes
Name	Required.
SKU	Your item code.
Category, Sub-category	Codes from your category tree.
Status	`active` or `inactive`. Blank means active.
Item type	The label or key of one of your item types, such as `finished_product`.
List price, Cost price, Tax %	Numbers.
Cost price as of	A date.
Quantity breaks	One cell: `from:rate; from:rate`, for example `100:390; 500:360`. A blank rate means that quantity is priced normally.
Unit of measure, Lead time (days), Description	
Supplier	The name of an existing supplier. Optional.

   A sample, saved as `.csv`:

```
Name,SKU,Category,Sub-category,Status,Item type,List price,Tax %,Unit of measure,Quantity breaks,Description
22 kW IE3 induction motor,MOT-22-IE3,Motors,Induction,active,finished_product,18500,5,Nos,5:17800; 10:17200,"415 V, 4 pole, B3 foot mount"
Deep groove ball bearing 6310-2RS,BRG-6310-2RS,Bearings,Ball,active,finished_product,64,5,Nos,,"50 x 110 x 27 mm, sealed"
```

3. **Avoid duplicates before you load.** Products Import does not check for an existing SKU or name. Load the catalogue once only. If some products already exist, press **Export** for Products, compare the SKUs with your sheet, and take the existing ones out of the import file. To change prices of products that exist, use Export and Update instead (Chapter 6, section 5).
4. **Split the file** into five files of 1,000 rows: `products-1.csv` to `products-5.csv`. There is no row limit, but the whole file is processed in one request. Smaller files are easier to check and to repeat.
5. **Load a small test first.** Take the first 10 rows into `products-test.csv`. Import it. On **Match columns**, check that every column points at the field you mean. On **Review**, read "10 ready". Press **Import 10 rows**. Open **Products** and check them: price, category, quantity breaks. If something is wrong, correct the products with Export and Update, and fix the sheet. Then take the 10 rows out of file 1, because they are already in.

6. **Load the five files, one by one.** For each: **Upload**, **Match columns** (keep the header names the same in every file, so the columns match the same way), **Review**, **Import N rows**. Read the **Result**. Download the report. A row with a problem, such as a category code that does not exist, is left out and listed with the reason. Fix those rows in a new file, `products-fix.csv`, and import that.
7. **Never import the same file twice.** If a load was interrupted, check **Products** first, to see how far it got.

Check it worked.

1. The five **Result** reports add up to 5,000 imported rows, plus any you fixed later. Add the 10 from the test.
2. Count through the API:

```
curl -s -H "Authorization: Bearer $KEY" "$BASE/products?count=only&aggregate=count,sum:list_price"
```

   `meta.total` is your count of products (plus any that existed before). `meta.aggregates.sum_list_price` should equal the sum of the List price column in your sheets.
3. Spot check one by SKU: `curl -s -H "Authorization: Bearer $KEY" "$BASE/products?search=MOT-22-IE3"`. The price, `qty_breaks` and `item_type` match your sheet. The cost price is not in the API. Check it with an Export of Products.
4. **Change history** lists the creates, with your name.

Common questions

1. Why do I get `404` on an endpoint that is in this guide? Either your workspace does not own the module behind it, or the id is not one in your workspace. A module that is off answers exactly like an address that does not exist. Ask your BPMSquare contact which modules your workspace has (Chapter 1, section 5).

2. Can I create or change accounts, products or invoices through the API? No. Quotations are the only object you can create, change or delete through the API. The other objects are read-only there. To load or correct them in bulk, use Data Workbench (Chapter 6). The one other write is raising a draft project invoice from worked hours.

3. A key that worked yesterday now answers `401`. What happened? The key was revoked, it passed its expiry date, or it is the workspace key and someone regenerated it. A scoped key whose date ran out behaves like a key that was never made. Look at the key list under **Settings > Integrations > API keys & developer**. Make a new key, and give your integration the new value.

4. Should I use the change feed or a webhook? Use the change feed when you want to set the pace, when you cannot offer a public address, or when you need the news within seconds. Use an Event webhook when you want BPMSquare to prompt you, and you are happy with batches on a schedule. Many integrations use both: the webhook says "something changed", and the feed is read for exactly what (Chapter 3).

5. Why did a webhook arrive late, or twice? Webhooks are delivered in batches on a schedule, not at the moment of the change. A batch is sent again if your receiver did not answer with a success, and each repeat has a new `delivery_id`. Keep the event `id` of what you handled, and skip repeats (Chapter 3, section 5).

6. Why does a change show `[redacted]`? Phone, email and tax number of accounts and contacts, and staff phone and email, are protected. The change feed and webhooks show that a change happened but not the value. Read the record with its endpoint if your key has the scope to see it.

7. An awarded quote did not reach Dynamics 365. Where do I look? Open **Settings > Integrations > Integration middleware** and the **Monitor** tab. Set the status chip to **Failed**. Click the arrow next to the status to read the reason and the **What to do** line. Fix the cause, tick the line, and press **Restart**. A quote never lands twice. If there is no line at all, check that the Standard Quote link is on under **Links**, that the quote's status is Accepted (awarded), and that the customer was read in from Dynamics (Chapter 5).

8. Can I run a Data Workbench import from a script? No. Data Workbench is a screen that signs in as a person. A script cannot call it with an API key. For scheduled loads of quotations, use the REST API. For other objects, a person runs Data Workbench, or your BPMSquare contact can advise on another way.