Matching payments to invoices via the API

When an integration posts bank vouchers (incoming payments) to PowerOffice Go via the API, you typically want the payment to be automatically linked to the correct invoice in the customer subledger. Go supports three different methods for achieving this, and the choice of method has concrete consequences for the integration workflow.

This article explains how the three matching methods work, when to use each of them, and the pitfalls you should be aware of.

TL;DR

Method	Requires	Matching timing	Recommended for
`CustomMatchingReference`	A custom reference set on both invoice and payment	Immediately at posting	Default use — flexible, does not require the invoice to be posted first
`InvoiceNo`	The invoice must already be posted in Go, and you must know its invoice number	Immediately at posting	When you can synchronize the invoice number back from Go before the payment is sent in
`Cid` (KID)	KID on both invoice and payment	Bulk (runs periodically, not in real time)	End-user payments coming in via OCR/direct debit files

> **Bottom line:** `CustomMatchingReference` is the recommended method for most integrations. It requires no synchronization of invoice numbers between Go and the source system — as long as you use the same reference on the invoice and the corresponding payment.

Fields on BankVoucherLinePostDto

When you post a bank voucher via `POST /v2/Vouchers/BankJournals`, each line (`BankVoucherLinePostDto`) has three fields relevant for matching against subledger accounts:

```
{
  "AccountId": 12345,
  "PostingDate": "2026-05-12",
  "CurrencyAmount": -1500.00,
  "VatId": null,
  "InvoiceNo": "100000187",
  "Cid": "00000302141000001872",
  "CustomMatchingReference": "ORDER-48130"
}
```

`AccountId` must point to a subledger account (typically customer receivables) for matching to take effect. If none of the matching fields are set, the payment will remain as an open entry on the customer ledger until matched manually.

The matching methods in detail

1. CustomMatchingReference (recommended)

A custom text string set on both the invoice and the payment. Go matches immediately at posting: when a voucher is posted with a reference, the system searches for the *oldest unmatched* entry on the subledger account with the same reference and opposite amount, and links them.

Advantages:

- Works regardless of order. It does not matter whether the invoice or the payment is posted first — as soon as the other voucher arrives with the same reference, they are linked.
- No need to synchronize Go-generated invoice numbers back to the source system.
- Matching happens in real time at posting.

Pitfall — the FIFO principle:

If you use the *same* `CustomMatchingReference` value on multiple vouchers, Go will match according to "first-in-first-out". This means that the oldest unmatched voucher with the reference will always be picked first — even if it is not necessarily the one you intuitively expected to be linked.

Example of the problem:

Date	Voucher	Type	Reference	Result
18.03	Invoice A	Outgoing invoice	`48130`	Pending
19.03	Invoice B	Outgoing invoice	`48130`	Pending
20.03	Payment 1	Bank voucher	`48130`	Matched against **Invoice A**
21.03	Invoice C	Outgoing invoice	`48130`	Pending
22.03	Payment 2	Bank voucher	`48130`	Matched against **Invoice B**

If "Payment 1" was actually meant for Invoice B, the linkage is wrong — even though the per-customer balances still add up.

Best practice: Use a *unique* `CustomMatchingReference` per invoice/payment pair. Typical values are order numbers, invoice numbers from the source system, or a GUID. That way you avoid the FIFO problem entirely.

```
// Invoice
{
  "CustomMatchingReference": "TREBYGG-2026-00187"
}

// Corresponding payment
{
  "CustomMatchingReference": "TREBYGG-2026-00187"
}
```

2. InvoiceNo (direct invoice matching)

Sets the invoice number directly on the bank voucher line. Go looks up the specific invoice in the subledger and links the payment directly to it — regardless of order or any other open entries.

Advantage: The most deterministic method. Guaranteed correct linkage no matter what is open in the ledger.

Limitation: The invoice must *already be posted in Go* by the time you submit the payment. In practice this means the integration must:

1. Send the invoice via the API
2. Capture the Go-generated invoice number from the response
3. Store the invoice number in the source system
4. Use the same invoice number when the payment is later submitted

If you cannot guarantee ordering — for example if payments may arrive before the invoice is booked — matching will fail. In practice many integrations instead use an external reference system via `CustomMatchingReference`.

```
{
  "AccountId": 12345,
  "InvoiceNo": "100000187",
  "CurrencyAmount": -1500.00,
  "PostingDate": "2026-05-12"
}
```

3. Cid (KID) — for incoming OCR payments

KID matching differs from the other two methods in that it does not run immediately at posting, but rather in a separate bulk process. This reflects the fact that KID is typically used for incoming bank payments (OCR files, direct debit) that are imported in batches.

Use this when:

- The invoices have KID identifying them
- Payments arrive via OCR/bank imports

Be aware that:

- Matching does not happen in real time — it can take some time before the link is in place
- Suited primarily for OCR/bank flows, not for direct API integrations that post payments manually

Which method should you choose?

Use this prioritization:

1. **Default recommendation:** Use `CustomMatchingReference` with a *unique* value per invoice/payment pair. This covers most integration needs and requires no synchronization of invoice numbers between Go and the source system.
2. **If the integration knows the Go invoice number at the time of payment** and wants maximum deterministic linkage: use `InvoiceNo`. Requires the invoice to be booked first.
3. **If the payment arrives via an OCR/bank file:** use `Cid`. Matching is then handled by the OCR bulk process.

What happens if multiple matching fields are set?

The following prioritization has been **empirically verified** through testing in the demo environment:

1. `InvoiceNo` — validated *synchronously* at POST time. The value must point to an open subledger entry on the same account. If it is invalid (points to a nonexistent or already matched invoice), the entire voucher is rejected with HTTP 404 — there is *no* graceful fallback to `CustomMatchingReference`. If the value is valid, the line is matched synchronously by the same worker that books the voucher.

2. `CustomMatchingReference` — matched synchronously by the same worker, but *only* if `InvoiceNo` is not set on the line. Requires an open subledger entry on the same account with opposite amount and the same CMR.
3. `Cid` **(KID)** — matched by a *separate asynchronous bulk process* that runs on an interval (observed at >5 minutes, likely on an hours/days scale). The bulk process does not evaluate lines that have already been matched via `InvoiceNo` or `CustomMatchingReference`.

> **Practical implication:** If you set both `InvoiceNo` and `CustomMatchingReference` on the same line "just to be safe", and `InvoiceNo` happens to be invalid, the entire voucher will be rejected — `CustomMatchingReference` will not save you. Set only one of the fields per line, and fall back to `CustomMatchingReference` only if you are unsure whether the invoice is booked yet.

Technical details from empirical testing

The following observations were made when testing against the demo environment and are useful to know when building the integration:

Posting is asynchronous

`POST /v2/Vouchers/BankJournals` (and the equivalent voucher endpoints) returns `201 Created` immediately, but the response contains a *negative placeholder* `VoucherNo`. The actual worker assigns the final voucher number within seconds. If you need the final voucher number, perform a `GET` against the voucher `Id` shortly after — or build a retry loop that waits until `VoucherNo > 0`.

InvoiceNo is validated only against open entries, not history

An invoice that has already been matched (paid) *cannot* be used as the `InvoiceNo` target in a new voucher. Attempts to do so return HTTP 404, the same as if the invoice had never existed. This has consequences for corrections/reversals: if you reverse a payment and later want to repost it, the invoice must first be open again.

Sign convention for incoming payments

For a payment to match against an outgoing invoice, the lines must have the *opposite* sign of the invoice on the subledger account. In practice: on an incoming payment the bank line must be *positive* (+) and the customer receivables line *negative* (−). The opposite sign creates a voucher that technically books but does not produce matching effects because it lands on the "wrong side" of the subledger balance.

Subledger AccountId for the customer must be retrieved correctly

When you set `AccountId` on a bank voucher line intended to match against a customer, the value must be the customer's *subledger account id* — not the customer's `CustomerId` or customer number. Retrieve the correct value from the `Customer.SubledgerAccountId` field via `GET /v2/Customers/{id}`. Using the wrong id results in either a 400 validation error or (worse) a posting on the wrong account that does not match.

Cleaning up incorrectly matched entries

If an integration has matched vouchers incorrectly (typically due to the FIFO problem with a reused `CustomMatchingReference`), this can be cleaned up manually in Go:

Ledger → Customer → Match entries — remove the existing link and re-match with the correct linkage.

Per-customer balances are not affected by incorrect matching — it is only *which* specific vouchers are linked together that is shifted. If the balance is what matters most to the customer, you can leave historical links as they are and focus on getting the flow right going forward.

Example: A complete bank voucher with matching

Here is a complete example of an incoming payment matched directly against an outgoing invoice via `CustomMatchingReference`:

```
POST /v2/Vouchers/BankJournals
Content-Type: application/json
Authorization: Bearer {token}
Ocp-Apim-Subscription-Key: {subscription-key}

{
  "VoucherDate": "2026-05-12",
  "CurrencyCode": "NOK",
  "Description": "Incoming payment, invoice 100000187",
  "BankVoucherLines": [
    {
      "AccountId": 11500001,
      "PostingDate": "2026-05-12",
      "CurrencyAmount": -54999.00,
      "VatId": null,
      "CustomMatchingReference": "TREBYGG-2026-00187"
    },
    {
      "AccountId": 19200001,
      "PostingDate": "2026-05-12",
      "CurrencyAmount": 54999.00,
      "VatId": null
    }
  ]
}
```

The invoice posted earlier must have had the same `CustomMatchingReference` on its customer receivables line for matching to take effect immediately.

Summary

- `CustomMatchingReference` is the recommended default method. It works in both directions (invoice before payment or vice versa) and requires no synchronization of Go invoice numbers.
- *The value must be unique per invoice/payment pair.* Reusing the same reference leads to FIFO matching, which can produce incorrect linkages.
- `InvoiceNo` provides deterministic linkage but requires the invoice to be booked and the integration to know the invoice number.
- `Cid` is used primarily for OCR/bank-file incoming payments and runs as a bulk process.
- Incorrectly matched history can be cleaned up manually in Go without affecting balances.

For questions about the matching flow in a specific integration, contact **go-api@poweroffice.no**.