



PARTNER DEVELOPER GUIDE

Integrating a Zid Store with an **ERP** or Accounting System

What an ERP, accounting, WMS or backend integration needs from the Zid platform – in the order you need it.

VERSION

1.2.0

LAST REVIEWED

7 October 2026

AUDIENCE

Partner developers &
solution architects

LATEST VERSION

docs.zid.sa

Document ID ZID-ZAM-DG-ERP • Classification: Partner distribution – may be shared with integration teams.

Verified against the Zid platform and a live development store. Not tax, legal, or compliance advice. If this copy is older than the online version, the online version applies.

START HERE

Read this page first

IN ONE SENTENCE

Zid pushes Orders, Products, Stock and Customers to you by webhook; everything else you poll inside a 60-requests/minute budget – so decide push-vs-pull per object before you write code.

Pick your reading path

ARCHITECT

Designing the integration

§1.1 Push vs pull → Part 2 Interface decisions → §3.6 Limits

DEVELOPER

Writing the connector

§1.2 Headers → §1.3 OAuth → §3.2 Orders → §3.5 Webhooks

FINANCE CONSULTANT

Mapping money & VAT

§1.6 Money → §1.10 Discounts → §3.7 Invoicing

Five mistakes that break integrations in month three

- 1 Putting `access_token` in the Bearer header – it goes in `X-Manager-Token`. [§1.2](#)
- 2 No monitored refresh job for tokens that expire after one year. [§1.3](#)
- 3 Recomputing totals instead of posting `payment.invoice[]`. [§1.10](#)
- 4 Triggering invoice logic off Order Status instead of payment status. [§3.7](#)
- 5 Retrying non-idempotent writes without searching first. [§3.5](#)

How to read the markers

WATCH OUT

Easy to get wrong and expensive to discover in production.

NOTE

Changes a design decision but will not break a working build.

EXAMPLE

A real observed payload with test values.

KSA ONLY Applies to stores in Saudi Arabia only – chiefly ZATCA e-invoicing and the national short address.

Document control

Document ID	ZID-ZAM-DG-ERP	Classification	Partner distribution
Version	1.2.0	Last reviewed	7 October 2026
Owner	Zid App Market – Partner Technical Support		
Source of truth	docs.zid.sa – the API reference is authoritative for every request and response schema		
Feedback & escalation	Partner Help Center • Zid developer Slack • Partner Dashboard • see §3.8		

CONTENTS

Contents

INTRODUCTION BEFORE YOU BEGIN

4

About this guide	4
Before you start	5

PART 1 CONCEPTS YOU NEED BEFORE WRITING CODE

6

1.1 Push vs pull: the one decision that shapes everything	7
1.2 Base URLs, API families, and request headers	9
1.3 Authorization: OAuth 2.0, tokens, and lifecycle	10
1.4 The install journey Zid requires	12
1.5 Scopes	13
1.6 Money: one model, full precision in API, two decimals on screen	15
1.6a Currency: order currency vs store currency	17
1.7 Order Status is the contract	18
1.8 Products, variants, stock	19
1.9 Customers: phone is the identity key	20
1.10 Discounts: where each mechanism appears	21
1.11 Customer wallets and cashback	24
1.12 Marketplace orders (Mazeed, Trendyol): identify, invoice, return	26

PART 2 INTERFACE DESIGN DECISIONS

28

2.1 Decisions to settle per data object	29
2.2 Mapping Zid concepts onto any ERP	30

PART 3 BUILDING IT

31

3.1 Products and stock sync	32
3.2 Orders: list, view, create	33
3.2a Order field reference	35
3.3 Returns, Credit Notes, refunds	40
3.4 Coupons and Bundle Offers	41
3.5 Webhook receiver	42
3.6 Errors, limits, silent incompleteness	44
3.7 Money, VAT, invoicing, and reconciliation	45
3.8 Testing, cutover, go-live	48

APPENDIX WORKED EXAMPLES AND CROSS-REFERENCE

50

A.1 Worked examples	51
A.2 Cross-reference: concern → section	52

INTRODUCTION

About this guide

KEY TAKEAWAY

A task-oriented companion to the API reference: which endpoints an ERP integration needs, in what order, and which behaviors break integrations in month three.

Audience. Developers, solution architects, and functional consultants connecting one or more Zid stores to an ERP, accounting system, WMS, or internal backend — as a private app (one store) or a public App Market app (many stores). The guide is system-neutral: it describes Zid's side of the contract and the decisions any ERP forces, not one vendor's connector.

What this guide is. A task-oriented companion to the [API reference](#). The reference tells you what a single endpoint accepts and returns; this guide tells you which endpoints an ERP integration needs, in what order, and which behaviors break integrations in month three. Everything here was verified against the Zid platform and reproduced against a live development store. Example values come from real test orders and are labeled as examples — your store's IDs, codes, and totals will differ; the shapes will not.

What this guide is not. Tax, legal, or compliance advice. VAT and invoicing sections describe platform configuration and field semantics only.

Markets. Zid stores operate in several countries, and the platform is not Saudi-only. Rules that apply **only to Saudi Arabia** are marked **KSA ONLY** throughout — chiefly e-invoicing (ZATCA/EGS, [§3.7](#)) and the Saudi national short address. Everything else — money precision, order status, catalog, customers, discounts, wallets, webhooks — is market-independent. VAT itself is **not** a fixed 15%: it resolves per destination country ([§1.6](#)), and the worked examples in this guide happen to come from a Saudi store at 15%. If you are building for stores outside Saudi Arabia, read [§1.6](#) and [§1.6a](#) closely, and treat [§3.7](#)'s e-invoicing half as out of scope until a store tells you otherwise.

Terminology. Zid's own names are used throughout: Order, Order Status, Reverse Order, Credit Note, Inventory Location, Product Stock, Product Variant, Custom Option, Coupon, Bundle Offer, Customer Wallet. A parenthetical appears only where the Zid name does not convey the stock or accounting behavior (e.g. Digital Products — non-stock items that need no shipping).

INTRODUCTION

Before you start

KEY TAKEAWAY

Read these four reference pages first. This guide assumes them and does not repeat them.

Page	Why it matters to an ERP integration
Start Here and Create your First App	Partner account, app registration, and how to get a development store to build against
Authorization	The OAuth flow and the two tokens every call needs
Zid App Activation & OAuth Policy	Mandatory install-flow rules. An ERP app that ignores these fails review – see §1.4
Rate Limiting	60 requests/minute per app per store. This number shapes your whole sync design

Also worth knowing before you build: Zid publishes official [SDKs](#), an [MCP server](#) and an [AI agent skill](#); if your app needs a merchant-facing settings screen (connection details, field mapping), build it as an [Embedded App](#) with [Zid MUI](#) rather than as an external portal.

How to use this guide as a knowledge base

Each section answers one question and states exactly where its data lives (endpoint + field path). If you are building an AI assistant, RAG index, or support playbook on top of this guide, treat each subsection as one retrievable unit: it contains the claim, the field paths, and one example. A cross-reference table at the end maps every ERP concern to its section.

Conventions used in this guide:

- **Every endpoint links to its reference page on first mention.** The reference page is authoritative for the complete request and response schema; this guide carries only the fields that change integration design.
- `GET /v1/managers/store/orders` means `https://api.zid.sa` + that path. Base paths and required headers differ by API family – see [§1.2](#).
- Money fields: compute on `value` (full precision), display `value_string`. Never parse display strings.
- Status and type codes are lowercase (`new`, `preparing`, `indelivery`, `product`, `coupon`). Display names are localized and must never drive logic.
- **Watch out** boxes mark behavior that is easy to get wrong. **Example** blocks show real observed payloads with test values.
- A few behaviors in this guide are real but do not yet have a reference page of their own; those are marked (no reference page yet) so you know the link is missing rather than hidden.

01

PART 1

CONCEPTS YOU NEED BEFORE WRITING CODE

1.1 Push vs pull: the one decision that shapes everything

1.2 Base URLs, API families, and request headers

1.3 Authorization: OAuth 2.0, tokens, and lifecycle

1.4 The install journey Zid requires

1.5 Scopes

1.6 Money: one model, full precision in API, two decimals on screen

1.6a Currency: order currency vs store currency

1.7 Order Status is the contract

1.8 Products, variants, stock

1.9 Customers: phone is the identity key

1.10 Discounts: where each mechanism appears

1.11 Customer wallets and cashback

1.12 Marketplace orders (Mazeed, Trendyol): identify, invoice, return

PART 1 • CONCEPTS

1.1 Push vs pull: the one decision that shapes everything

KEY TAKEAWAY

For every object your integration touches, answer one question: does Zid push changes to you (webhook), or must you poll (API)? Settle this map before sizing anything.

Objects with webhook events run near-real-time at low request cost. Objects without events must be polled out of a capped budget (60 requests/minute per app per store).

Object	Webhook events	Merchant API	Integration pattern
Order	<code>order.create</code> , <code>order.status.update</code> , <code>order.payment.status.update</code> , <code>order.tagged</code> (POS only), <code>order.refund.create</code> , <code>payment.failed</code>	Full (<code>list</code> , <code>view</code> , <code>create</code> , <code>status</code> , <code>tags</code> , <code>comments</code>)	Event-driven + <code>updated_at</code> polling as backstop
Product	<code>product.create</code> , <code>product.update</code> , <code>product.publish</code> , <code>product.delete</code>	Full (Products family headers)	Event-driven
Product Stock	<code>product.stock.create</code> , <code>product.stock.update</code> , <code>product.stock.delete</code>	Full + bulk	Event-driven; bulk for initial load
Inventory Location	<code>location.create</code> , <code>location.update</code> , <code>location.delete</code> , <code>location.shipping-zone.attach</code> , <code>location.shipping-zone.detach</code>	Full	Event-driven, rarely polled
Product Category	<code>category.create</code> , <code>category.update</code> , <code>category.delete</code>	Full	Event-driven
Customer	<code>customer.create</code> , <code>customer.update</code> , <code>customer.merchant.update</code> , <code>customer.login</code>	Full (<code>list</code> , <code>view</code> , <code>profile</code> , <code>create</code> , <code>update</code> , <code>tags</code> , <code>bulk</code> , <code>import</code>)	Event-driven
Abandoned Cart	<code>abandoned_cart.created</code> , <code>abandoned_cart.completed</code>	Read + recovery messaging	Event-driven
Reverse Order (returns)	— none —	Full lifecycle (<code>view</code> , <code>calculate</code> , <code>create</code> , <code>waybill</code> , <code>classify</code> , <code>refund</code> , <code>reasons</code>)	Poll: trigger off Order reaching <code>reverse_in_progress</code> , or off <code>order.refund.create</code>
Credit Note	— none as standalone event —	Read per Order	Poll: trigger off Order status change; read after returns, cancellations, edits
Coupon	— none —	Full (<code>list</code> , <code>view</code> , <code>create</code> , <code>update</code> , <code>delete</code> , <code>bulk</code>)	Poll on your own schedule
Bundle Offer	— none —	Full (<code>list</code> , <code>view</code> , <code>create</code> , <code>update</code> , <code>delete</code>)	Poll; read before order reconciliation
Customer Wallet, Cashback	— none —	Full (<code>balance</code> , <code>credit</code> , <code>debit</code> , <code>bulk</code> , <code>transactions</code> , <code>cashback rules</code>)	Poll; treat as money movements, never as a mirrored balance — see §1.11

Object	Webhook events	Merchant API	Integration pattern
Loyalty	– none –	Read configuration, manage rules	Read for reconciliation; never synchronize balances
Shipping methods, VAT settings, store identity, brands, tags	– none –	Read (shipping, VAT, store, brands)	Cache at startup; re-read periodically (no change events exist)

Exact event payloads and condition syntax are on the [Webhooks overview](#) and the per-object event pages: [Order](#), [Product](#), [Customer](#), [Product Category](#), [Abandoned Cart](#).

WATCH OUT

Two events are easy to miss and matter to finance: `order.refund.create` (the only push signal anywhere in the returns flow) and `payment.failed`. Subscribe to both. `order.tagged` exists but fires only from POS, not from Merchant API tag writes – do not use it to confirm your own tag calls.

PART 1 • CONCEPTS

1.2 Base URLs, API families, and request headers

KEY TAKEAWAY

One store gives you one token pair, but each API family expects it in different headers. Build your HTTP client to take the family as a parameter, and never hand a request builder for one family to another.

This is the most expensive thing to get wrong, because the symptom is a `401` or `403` on a valid token.

API family	Base path	Required headers
Orders, Reverse Orders, Customers, Coupons, Bundle Offers, Loyalty, Abandoned Carts, Store Settings, Webhooks, App Management	<code>/v1/managers/...</code>	<code>Authorization: Bearer <authorization></code> <code>X-Manager-Token: <access_token></code>
Products, Categories, Attributes, Product Stock, Inventory Locations	<code>/v1/products/</code> , <code>/v1/...</code>	<code>Access-Token: <access_token></code> <code>Store-Id: <numeric id></code> <code>Role: Manager</code>
Customer Wallets, Cashback Rules	<code>/v1/customer-wallets/...</code>	<code>X-Manager-Token: <access_token></code> <code>Store-Id: <numeric id></code>

All base paths are under `https://api.zid.sa`. Send `Accept: application/json` on every call in every family.

Verify a new token pair

Call `GET /v1/managers/store/reverse-orders/reasons` and assert HTTP `200` with `Content-Type: application/json`. Without an `Accept: application/json` header, an unauthenticated call returns a `302` redirect to a login page, which HTTP clients with redirect-following turn into a misleading `200` HTML page. Always assert content type, not just status.

```
curl https://api.zid.sa/v1/managers/store/reverse-orders/reasons \
  -H "Authorization: Bearer $AUTHORIZATION" \
  -H "X-Manager-Token: $MANAGER_TOKEN" \
  -H "Accept: application/json" # without it, a 401 becomes 302 → HTML 200
```

CURL

Rules that never change

1. The field named `access_token` in the OAuth token response is not the bearer token. It carries store context and goes in `X-Manager-Token` or `Access-Token`. The field named `authorization` is the bearer token. Store them under distinct names (`authorization_token`, `manager_token`) with a comment recording which response field each came from. This single confusion causes more partner `401`s than anything else in the API.
2. `Store-Id` is the numeric store ID. The store UUID is a different identifier that also appears in payloads – keep both, never substitute.
3. On the Products family, `X-Manager-Token` + `Store-Id` is accepted as an alternative to `Access-Token` + `Role: Manager`, and selects the manager role by itself. `Access-Token` without `Role: Manager` is treated as a customer-role request and will fail or return storefront-shaped data – if a Products call returns less than you expect, check `Role` before anything else.
4. A client built from the Orders contract returns `401` on every Products call and vice versa. If you get `401` with a fresh token, check the family first, the token second.
5. Response envelopes and error shapes differ by family – see Responses and §3.6.

PART 1 • CONCEPTS

1.3 Authorization: OAuth 2.0, tokens, and lifecycle

KEY TAKEAWAY

Use the authorization code grant, validate `state`, exchange the code server-side, persist all four token fields with an `issued_at`, and run a monitored refresh job long before the one-year expiry.

Full flow and sample code: [Authorization](#). The security requirements below are mandatory under the [Activation & OAuth Policy](#) — they are review criteria, not suggestions.

OAuth 2.0 authorization code grant

Required for App Market apps; works for private apps too.

1. **Generate a `state` value** — random, one-time, short-lived — bind it to the initiating session, and store it server-side.
2. **Authorize:** GET `https://oauth.zid.sa/oauth/authorize?client_id=<id>&redirect_uri=<exact_registered_url>&response_type=code&state=<state>`. The redirect URI must match the registered value exactly. Use PKCE where your client configuration supports it, and wherever Zid requires it.
3. **On the callback, validate `state` before doing anything else** and reject the request if it does not match the initiating session. Never log the authorization code, and never forward it into another URL.
4. **Exchange promptly and server-side** (a server component is mandatory): POST `https://oauth.zid.sa/oauth/token`, form-encoded, with `grant_type=authorization_code`, `client_id`, `client_secret`, `redirect_uri`, `code`. Do not hold a code while you provision anything — exchange first, provision after.
5. **The response contains `access_token`, `authorization`, `refresh_token`, `expires_in`**. Persist all four plus an `issued_at` timestamp immediately. Keep client secrets and tokens server-side only — never in browsers, mobile apps, client-side storage, logs, analytics, or repositories.
6. **Refresh before expiry** with `grant_type=refresh_token`.

NOTE

Zid Bridge (`https://bridge.zid.dev`) walks the flow in a browser and hands you tokens without implementing callbacks. It is supported for testing and private apps only — never for an App Market app's install flow, where the policy requires OAuth to begin when the merchant selects Activate and explicitly forbids asking a merchant to paste tokens or type a Store ID. Expiry still applies: Bridge removes implementation work, not the renewal obligation.

Token lifecycle: the year-one outage

The authorization token, the manager token, and the refresh token each live one year. Long lifetimes make the failure mode silence: the integration runs correctly for months and stops on a day nobody is watching. Refresh well before the deadline — around the ten-month mark — and key the job off the `expires_in` you were actually issued plus your stored `issued_at` rather than a hard-coded year.

Three controls, not one:

1. **A scheduled refresh job** keyed on stored `issued_at` and `expires_in`, refreshing anything inside its last 30–60 days.
2. **An alert on refresh failure** — an unmonitored refresh job is a second silent failure.
3. **A named owner** in the runbook.

Zid also emails the partner about tokens approaching expiry – once at 15 days out, then daily through the final 14 days. Treat that as a safety net, not a control: it reaches a mailbox, not your scheduler, and it cannot act on your behalf.

On `401`, attempt one refresh-and-retry; if that fails, surface an explicit reauthorization-required state. Never loop retries against `401`. Your app must survive token renewal, reauthorization, reinstall, and permission changes without creating duplicate or cross-store connections.

On uninstall (`app.market.application.uninstall`, see [App Management events](#)), every token for that store is revoked – both the OAuth access and refresh tokens and the manager token: mark tokens revoked, stop jobs, stop retrying, and stop processing that store's data.

WATCH OUT

Uninstall is not the only way a working token dies. A store that loses its plan subscription or its API eligibility can have its tokens revoked administratively, with no uninstall event delivered to you. So a sudden `401` on a store that worked yesterday has three causes, in order of likelihood: expiry, an uninstall you missed, and lost eligibility. Surface a reauthorization-required state and tell the merchant which one it is – do not retry, and do not assume a missing uninstall event means the install is still live.

PART 1 • CONCEPTS

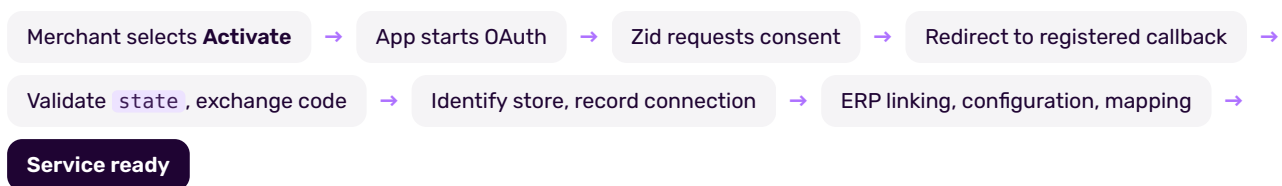
1.4 The install journey Zid requires

KEY TAKEAWAY

OAuth comes first – before any partner registration, ERP credentials, or tenant provisioning. Report "Service ready" only once setup is complete and the service can actually operate.

An ERP integration is the hardest case for Zid's [Activation & OAuth Policy](#), because the app genuinely cannot sync anything until the merchant supplies ERP connection details and a field mapping. Read the policy before you design the first screen – the common ERP-vendor install flow (collect ERP credentials, provision a tenant, then connect Zid) is the exact pattern the policy rejects unless you have prior written approval.

The standard journey



What this means in practice for an ERP app

- **OAuth comes first.** Partner registration or login must not precede it.
- **The callback must process the OAuth result and continue the journey.** Landing the merchant on a generic homepage, marketing page, or a settings area where they must hunt for the Zid integration is a rejection reason.
- **Never ask the merchant to paste** a Zid access token, refresh token, manager token, API key, client secret, or password, and never ask them to type a Store ID or store URL to establish the connection. The OAuth result tells you which store you are connected to.
- **Keep the verified store pinned** through signup, login and account linking. Do not attach a store to an existing ERP account just because an email address matches – require authenticated access and explicit merchant confirmation.
- **Do not start background synchronization** before required terms and setup are complete, and do not describe the service as ready while mandatory setup is outstanding.
- **If your product truly requires service prerequisites before OAuth**, that is the partner-managed setup model and it needs prior written approval from Zid. A free listing, no Zid plans, or external billing does not grant it.
- **If you ship a settings or mapping screen** inside the Zid dashboard, build it as an [Embedded App](#) and do not require a separate login for it.

Apps that do not comply may be returned for correction, rejected at review, suspended, or removed. If your app charges through Zid, plans and usage are handled by [App Management](#) – see [Subscription Details](#) and [Update Usage-Based Charges](#) – and any required app-plan payment completes through Zid before activation is available.

PART 1 • CONCEPTS

1.5 Scopes

KEY TAKEAWAY

Request only the scopes your interface uses. Scopes are configured on the app, not on the authorize URL – and any change after install means the merchant must authorize again.

Select scopes per app in the [Partner Dashboard](#); the merchant approves them at authorization. Request only what the interface uses – the policy requires it, and over-asking is a review finding.

Do not send a `scope` parameter on the authorize URL. Zid builds the consent request from the scopes configured on your app, and an app-level scope list is the only thing that decides what a token can do.

Who can change scopes, and when. While your app is not published (Draft, Rejected, or any other non-published status), you can select and change any available scope in the Partner Dashboard yourself, with no prior approval from Zid. Once the app is published, scope changes require Zid's approval.

Changing scopes after install forces re-authorization. The authorize endpoint skips the consent screen only when the store's existing valid token carries exactly the scope set your app is now configured for. Add or remove one scope and that match fails, so the merchant sees consent again. Until they complete it, the old token stays alive with its old scope set and every call needing the new scope returns `403`. Plan for this: ship a scope change as a prompted reconnect, not as a silent deploy, and detect it by `403` on a call you know is correct rather than by assuming your dashboard change took effect.

Scope	Grants
<code>third_create_order</code>	Validate and create Orders
<code>third_order_read /</code> <code>third_order_write</code>	Orders, Order Tags, order fulfillment inventories
<code>third_order_reverse_read</code>	Reverse Order reasons
<code>third_order_reverse_write</code>	Create/manage Reverse Orders, refunds, and Calculate reverse totals (a read-like call that needs write scope)
<code>third_abandoned_carts_read</code>	Abandoned Carts, recovery messages
<code>third_product_read /</code> <code>third_product_write</code>	Products
<code>third_product_stock_read /</code> <code>third_inventory_write</code>	Stock records, inventory writes
<code>third_customers_read /</code> <code>third_customers_write</code>	Customers, customer tags
<code>third_customer_wallet_read /</code> <code>third_customer_wallet_write</code>	Customer Wallets, cashback rules (no reference page yet)
<code>third_coupons_read /</code> <code>third_coupons_write</code>	Coupons
<code>third_bundle_offers_read /</code> <code>third_bundle_offers_write</code>	Bundle Offers (write exists – creation is supported)
<code>third_loyalty_read /</code> <code>third_loyalty_write</code>	Loyalty

Scope	Grants
<code>third_vat_read</code>	VAT configuration
<code>third_payment_read</code>	Payment-method display options
<code>third_countries_read</code>	Countries and cities
<code>third_one_signal_write</code>	OneSignal settings – also required to read them
<code>third_webhook_read</code> / <code>third_webhook_write</code>	List/register webhooks and health

Two patterns hold across the API: **write scope also grants read** (request either), and **reverse operations are asymmetric** – almost everything, including quoting a return, needs `..._write`.

NOTE

Scopes are not the only gate. A `403` on a correctly scoped call usually means one of three things: the merchant's plan does not include the feature, the store is outside the feature's supported market, or the manager lacks the permission (see [User Roles and Permissions](#)). Private-app eligibility is commercial, not technical: the store needs the API access add-on. An ineligible store cannot install the app, and debugging OAuth will not change that – verify eligibility before investigating anything else.

PART 1 • CONCEPTS

1.6 Money: one model, full precision in API, two decimals on screen

KEY TAKEAWAY

Compute on `value` at full precision, display `value_string`, parse every money field to a decimal type at the boundary, and read VAT settings per destination country instead of hard-coding a rate.

Every money figure in Orders, Credit Notes, and carts uses the same structure:

```
{"code": "total", "value": "314.96637500000000", "value_string": "314.97 SAR", "title": "Total"}
```

JSON

- Compute on `value` (up to 14 decimal places). Display `value_string`. Never parse display strings.
- Branch on `code`. Never branch on `title` (localized).
- The storefront shows 2 decimals (`138.00`), while the API carries full precision (`138.002875`). A catalog price of `120.0025` displays as `120.00` and totals as `138.00`, but the API line holds `120.0025` / `138.002875`. Post the API total into the ERP and add a rounding line if your ERP recomputes differently – never "correct" the source figure.

Product catalog prices accept 4 fractional digits (up to 15 digits total). Most ERPs store 2–3, so decide the rounding rule and direction once, and never write a rounded value back as if it were the source price.

JSON types: parse defensively, never trust the shape

The payload mixes JSON types in ways that are not predictable from a field's name, because different fields are backed by different column casts. Deserialize money to a decimal type at the boundary and coerce booleans, whatever JSON type arrived. A strict typed deserializer will fail on real payloads.

Pattern	Examples from one observed Order	What to do
Money as a JSON string with 14 decimals	<code>order_total: "314.96637500000000",</code> <code>payment.invoice[].value:</code> <code>"256.49250000000000",</code> <code>line_tax_amount:</code> <code>"18.00037500000000",</code> <code>net_sale_price:</code> <code>"79.99000000000000"</code>	Parse to decimal. Do not assume numeric
Money as a JSON number	<code>transaction_amount: 314.966375,</code> <code>line</code> <code>net_price: 99.99,</code> <code>total: 138.002875,</code> <code>discounted_tax_amount: 18.000375</code>	Parse to decimal too – <code>discounted_total:</code> <code>91.98849999999999</code> in the same payload is float dust, which is exactly why you must not keep these as floats
Booleans as integers <code>0 / 1</code>	<code>is_guest_customer: 1,</code> <code>has_different consignee: 0,</code> <code>is_gift_order:</code> <code>0,</code> <code>customer.verified: 1</code>	Coerce truthily; never <code>=== true</code>
Booleans as real booleans	<code>requires_shipping: true,</code> <code>is_taxable: true,</code> <code>is_discounted: false,</code> <code>is_potential_fraud:</code> <code>false,</code> <code>cod_confirmed: false</code>	Same coercion path, so one helper covers both
Empty collection as <code>[]</code> , populated as an object	<code>line_discounts: []</code> when nothing fired, <code>{coupon: {...}}</code> or <code>{discount_rule: {...}}</code> when something did	Treat an empty array as "no discounts". A schema that insists on an object breaks on every undiscounted line
Object-or-null, shape varies by level	<code>line_weight: {value: 2.5, unit: "kg"}</code> or <code>null</code> , while the Order's own <code>weight: 2500</code> is a bare number	Check the level you are reading before assuming a shape

WATCH OUT

created_at and **updated_at** are UTC. **issue_date** is not a timestamp at all — it is a localized display string ("05-10-2026 | 11:20 ص" : d-m-Y | h:i A , rendered in the store's timezone and in Arabic). Do not parse it, and do not diff it against **created_at** ; they differ by the store's UTC offset by design. Use **created_at** / **updated_at** for all date logic, and remember §3.2's date-only filters apply Asia/Riyadh day boundaries.

VAT: a fraction in logic, a percent in labels

tax_percentage is a fraction (0.15 on a Saudi store), and VAT is computed **per line after discount, then summed** — never on the header, never before discount. An ERP computing VAT header-side will drift by small persistent amounts. Display layers show the percent form (15 , VAT (%15) , applicable_tax_percentage: 15). Whether Zid prices are gross or net comes from **is_vat_included_in_product_price** in the **VAT settings** — read it before mapping anything (wrong here puts every document off by the VAT rate). That gross-vs-net flag is store-wide; the rate is not.

The rate is per destination country, not per store. Zid resolves tax from the Order's shipping-address country: a store can hold a tax row per country (rate, tax number, registration certificate), and any destination with no row of its own falls back to a single other-countries rate. So one store legitimately charges different rates to different destinations; a 0% country row is a real configuration and is not the same as having no row; and a store may have the tax feature switched off altogether. Never hard-code 15% or any other figure, never infer the rate from the store's own country, and never recompute a rate you were given — read it from the VAT settings, keyed by destination. Zid also models self-paid tax, where the merchant absorbs the VAT instead of adding it on top; that changes which figure your ERP treats as the taxable base.

PART 1 • CONCEPTS

1.6a Currency: order currency vs store currency

KEY TAKEAWAY

Every money figure on an Order is in the order currency (`currency_code`), not the store's. Record both currencies and both rates, and decide once which figure posts to the GL.

A Zid store has a base currency and can sell in others. After VAT, this is the most common source of wrong numbers in an ERP – and it is not optional reading for a store outside Saudi Arabia, or for any store with a multi-currency storefront.

- **An Order is denominated in the currency the buyer used.** `currency_code` on the Order is the order currency, and every money figure on the Order – line prices, `payment.invoice[]`, `order_total` – is in that currency, not the store's. Post with `currency_code` attached, or you will book the right number against the wrong unit.
- **Two currencies and two rates are recorded on every Order**, and the `currency` block (§3.2a) exposes both:

Block	Shape	Meaning
<code>currency.order_currency</code>	<code>{id, code, exchange_rate}</code>	What the buyer saw and paid in. All Order money is in this
<code>currency.order_store_currency</code>	<code>{id, code, exchange_rate}</code>	The merchant's base currency, with its rate at the time of the Order

- **`transaction_amount` is a derived figure, in the store currency.** When the two currencies differ, Zid computes `order_total × store_rate ÷ order_rate`. Both rates are stored on the Order, captured when it was placed, so the conversion is historical and reproducible – it will not drift when rates move later. Do not re-convert at today's rate, and do not expect `transaction_amount` to equal what the gateway actually settled to the bank.
- **Decide once, per deployment, which figure posts to the GL:** the order-currency total with its currency code, or the converted store-currency amount. Record both, plus both rates, so a later audit can reproduce the arithmetic.
- `order_total_string` is formatted in the order currency and `transaction_amount_string` in the store currency. As everywhere, display the strings and compute on the numbers (§1.6).
- **Catalog prices are held per store, not per country.** Multi-currency storefront pricing is a presentation-time conversion, so do not expect a per-currency ERP price list unless the merchant maintains one deliberately.

WATCH OUT

`has_different_transaction_currency` is narrower than it reads. It is `true` only when the payment method is ZidPay and the currencies differ. An Order paid by any other method in a non-store currency still has two different currencies while this flag stays `false`. Compare `currency_code` against `currency.order_store_currency.code` yourself – do not use the flag as your multi-currency test.

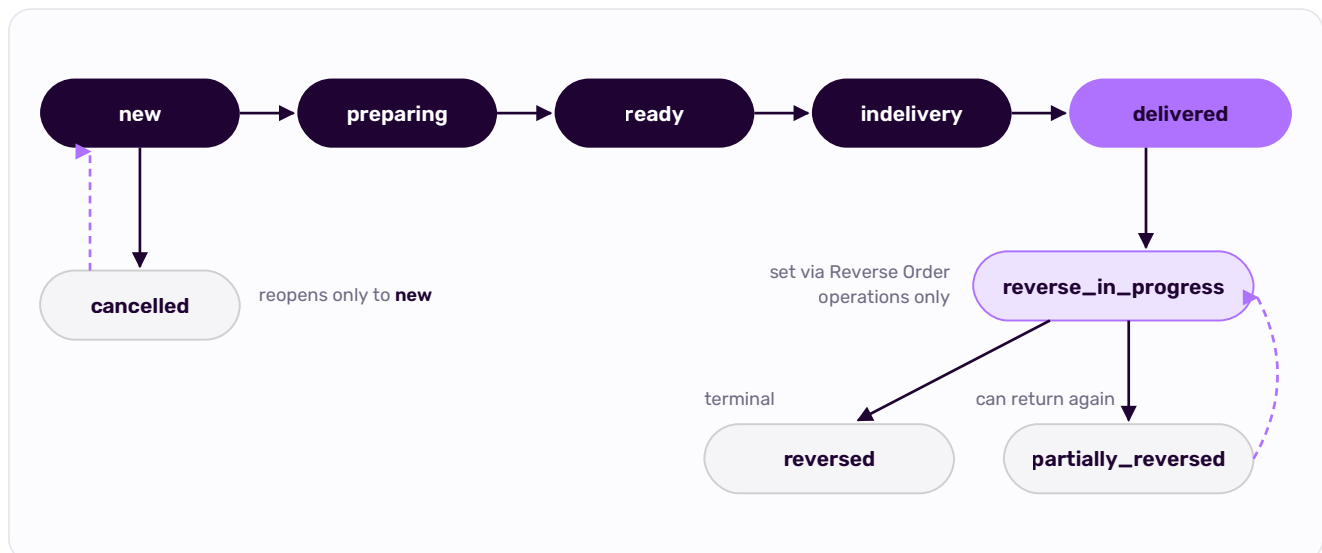
PART 1 • CONCEPTS

1.7 Order Status is the contract

KEY TAKEAWAY

Map your ERP document statuses onto Zid's set, not the reverse – some transitions cannot be forced through the API.

Change status with [Change Order Status](#).



- `reverse_in_progress` cannot be set directly – use the Reverse Order operations.
- `reversed` is terminal. `cancelled` normally reopens only to `new`.
- A return starts only from `delivered` or `partially_reversed`.
- Status spelling is lowercase everywhere (`indelivery` in both API and webhook conditions – normalize case-insensitively when matching).
- Custom sub-statuses exist per store ([list](#), [bulk create/update](#)). Never hard-code their codes; `reverse_in_progress` and `partially_reversed` are not customizable. **Custom sub-statuses emit no webhooks** – poll if the ERP depends on them.

PART 1 • CONCEPTS

1.8 Products, variants, stock

KEY TAKEAWAY

Decide the ERP item-type mapping before the first load, map variants (never parents), pick one stock model per product, and use the bulk endpoints — per-product sync of a large catalog takes hours at 60 requests/minute.

- **Product class is immutable.** Standard, grouped, voucher, downloadable, crowdfunding, donation, dynamic bundle (plus gifting): choosing wrong means recreating the product with a new UUID and remapping every cross-reference. Decide the ERP item-type mapping before the first load — see [Create Product Types](#). Digital classes (`downloadable` , `voucher`) are non-stock service items: the platform forces `quantity: null` , infinite stock, no shipping.
- **Variants are full pricing and inventory entities, not labels:** own price, sale price, cost, weight, and stock records. [Create variants](#) by selecting attributes (`[[slug, value]]`) — the slugs come from the store's [Product Attributes](#) and [Attribute Presets](#), so read or create those before you build the variant mapper. Combinations, SKUs, and non-empty barcodes must be unique. Omit `id` to create, supply it to update, `is_deleted: true` to remove. Creating variants on a standalone product converts it to a parent — one way. **Never map a parent product to an ERP item**; only its variants carry meaningful price and stock.
- **Stock has two mutually exclusive models:** simple quantity or per-location records — never both on one product. One record per product per location; `is_infinite: true` makes quantity meaningless. For volume, use [Bulk Update Product Stock](#) and [Update Product Stock for Location](#) (multiple products at one location atomically; multiple locations for one product), not per-product `PATCH` — at 60 requests/minute, per-product sync of a large catalog takes hours. Decide whether Zid receives **available** or **on-hand** quantity (sending on-hand oversells wherever the ERP holds reservations) and how infinite stock is represented.
- **Custom Option Fields** (buyer picks from up to 30 choices, [create](#)) and **Custom Input Fields** (buyer types/selects a value: text, number, URL, file, image, date, time, [create](#)) can both carry prices. Their contribution is called **additions** on the Order line: a line's value is price plus additions, times quantity, less discount, plus tax. Map only price × quantity and every configured line silently understates. On create they are keyed by `label` , not `name` .
- **Names and descriptions are language maps (`{ar, en}`)** — always send the map; a plain string risks leaving the other language blank on the storefront.
- **Listing:** [List Products](#) defaults `page_size` to 10 and caps it at 100 (larger values return 100 rows); follow `next` / `previous` links rather than assuming a size; order with `ordering` (`-` prefix descends; `-updated_at` for delta sync); deduplicate by product UUID while paging. Two filters take `true` only (`is_purchasable` , `has_weight` — `false` does not select the inverse; fetch and filter locally).
- **Writes:** `POST` returns `201` with the product — persist the UUID immediately. `PATCH` is partial. No write takes an idempotency key: on an unknown outcome, `GET` before repeating. `DELETE` returns `204` , soft-deletes, removes stocks, and cleans variants asynchronously. Unpublish (don't delete) products that must stay manageable, and mark the cross-reference row inactive rather than deleting it — historical Orders still need the mapping.
- **For the initial catalog load,** [Import Products via CSV/XLSX](#) and [Bulk Update Products](#) are usually cheaper than paging the API product by product.

PART 1 • CONCEPTS

1.9 Customers: phone is the identity key

KEY TAKEAWAY

Duplicate detection is by normalized phone, not email. Match customers on phone, and look the phone up before retrying any create.

- **Duplicate detection is by normalized phone, not email** – a conflict returns HTTP 409. Email is optional throughout, and payloads may contain non-deliverable placeholder addresses. Match customers on phone ([Find Customer by Phone Number](#)), never on email; look the phone up before retrying any create (creates carry no idempotency key).
- **Single create accepts mobile 8–14 digits; bulk create requires 9–14** – validate to the stricter bound.
- **Business (B2B) customers** add `type: business` with `business_name`, `tax_number`, `commercial_registration` – the fields a compliant tax invoice needs. These same fields appear on Order buyers.
- **Update is PUT** with `name` + `mobile` always required, but optional fields merge (omitted fields are preserved – except tags, see below). A null/zero city leaves the stored city unchanged.
- **Customer tags:** max ten per customer; `POST` adds one (find-or-create – repeating is safe); `PUT` **replaces the complete set** (an empty set clears all tags). If another system also tags, read-merge-write or you will silently delete its tags.
- **Bulk create (1–100 rows)** skips duplicates and returns only newly attached customers – a short response is normal; all-duplicate returns 409. `File import` returns a `job_id` proving acceptance, not completion (rows process in groups of 100; one group can fail while others succeed) – reconcile afterwards.
- **Deleting a customer removes the store relationship only.** Historical Orders keep their customer data, and erasure requests follow a separate approved process.
- **Pagination on List Customers:** `per_page` max 1000 (default 15); cursor mode (`after` , starting 0) is incompatible with custom ordering – use it for the initial load. `date_from` / `date_to` apply only together.

PART 1 • CONCEPTS

1.10 Discounts: where each mechanism appears

KEY TAKEAWAY

Post `payment.invoice[]` as the authoritative breakdown and use mechanism fields for reporting only. Recomputing a total from mechanism inputs is how integrations end up fractions out with no explanation.

Mechanism	List orders (simple / default)	View order	Invoice line codes (observed)
Coupon (one or many – always read the <code>coupons[]</code> array; the singular misses multi-coupon Orders)	absent / present	present	<code>coupon</code> (line discount), <code>sub_totals_after_coupon_discount</code> , <code>free_shipping_coupon</code> , <code>coupon_cod_discount</code>
Line / sale-price discount	absent / present (<code>discounted_total</code> , <code>discount_percentage</code> , <code>discounts</code>)	present	via line fields, no separate code
Bundle Offer (15%-off-one style)	absent / present (<code>bundle_products</code>)	present – see identification rule below	<code>products_discount</code> , <code>sub_totals_after_products_discount</code>
Loyalty redemption	absent / invoice line	invoice line	<code>loyalty_redemption</code> (plus <code>qitaf_redemption</code>)
Cashback	absent / present (<code>cashback</code> : number, object, or null)	present	– (reporting field)
Gift card	present even in simple (<code>gift_card_details</code>)	present	– (reporting field)
Free shipping	absent / invoice line	invoice line	<code>free_shipping_coupon</code> , <code>shipping_discount</code>
Progressive (tiered) discount	via coupon lines	via coupon lines	<code>coupon</code> subtype; automatic ones self-apply with an on-cart progress meter
Online-payment / card promotion	absent / present	present	<code>products_discount</code> , <code>sub_totals_after_products_discount</code> – the same codes as a Bundle Offer (see below)
Customer Wallet	absent everywhere	absent	cart-only mechanism – reconcile via <code>invoice[]</code> , never as an order field (see §1.11)

The full vocabulary of discount-side invoice codes is small and closed: `coupon`, `shipping_discount`, `free_shipping_coupon`, `coupon_cod_discount`, `products_discount`, `loyalty_redemption`, `qitaf_redemption`. Each is accompanied by a running-subtotal code (`sub_totals_after_coupon_discount`, `sub_totals_after_products_discount`, `sub_totals_after_loyalty_redemption_discount`). If you handle those, you handle every discount mechanism on the platform.

Only one discount may survive – unless the store has multi-discount enabled

Zid runs two different discount-combination modes, and which one a store is in changes the totals your ERP receives:

- **Multi-discount off (the default shape).** The engine computes the coupon discount, the automatic discount (bundle or online-payment rule) and the loyalty redemption, then keeps only the largest and removes the others from the cart. A coupon the customer entered can silently stop applying because a bundle beat it. The surviving mechanism is the one in `invoice[]`; the others leave no trace.
- **Multi-discount on.** Coupon, `products_discount` and `loyalty_redemption` can all appear on the same Order, each as its own invoice line.

It is controlled by a store setting (`multi_discounts_support`) together with a plan feature, so it varies per merchant and can change under you without any API event. Two consequences: never reconstruct "what the customer should have been charged" from the mechanisms you can see, and never assume a stored coupon applied just because it was attached to the cart. Post `invoice[]`. A free-shipping or free-COD coupon is the exception that survives alongside an automatic discount, which is why those codes can appear next to `products_discount`.

Identifying Bundle Offers on an Order

Two different questions, two different field groups – mixing them up is the most common bundle bug:

1. **Which offer is associated with a line?** Read `products[].meta.bundle_offer` → `{id, name, discount_percentage}`. It is present on every line the offer touches whether or not the discount fired. Use the `id` against [View Bundle Offer](#) to resolve that offer's conditions and dates. The same shape arrives in `order.status.update` webhook payloads.
2. **Did the discount actually fire?** All three must hold: (a) the line's `discounts` contains a `discount_rule` key (`{amount, percentage, unit_amount}`), (b) the line's `is_discounted` is `true`, (c) `payment.invoice[]` contains a `products_discount` line. If any is missing, the badge was present but no money moved – post the lines at face value. `coupon` / `coupons[]` stay empty on pure-bundle Orders, so a bundle never hides inside the coupon fields. Note that `discounts` arrives as `[]` on a line where nothing fired, and as an object where something did ([\\$1.6](#)).

EXAMPLE

Observed: a bundle that did not fire. A live 3-line Order carried `meta.bundle_offer` (`{id, name, discount_percentage: 15}`) on every line, yet each line had `is_discounted: false` and `discounts: []`, and `payment.invoice[]` held no `products_discount` – subtotal 256.4925, VAT 38.473875, shipping 15.00, COD 5.00, total 314.966375, with nothing discounted. The badge was associated with all three products; the rule's conditions were not met. **Posting a 15% reduction off the meta block alone would have under-billed the merchant on all three lines.**

WATCH OUT

`discount_rule` + `products_discount` does not by itself mean "bundle". Both mechanisms in Zid's automatic-discount engine – Bundle Offers and the online-payment/card promotion – produce the same `discount_rule` breakdown and the same `products_discount` invoice line. The only thing that separates them is question 1: a bundle carries `products[].meta.bundle_offer`, an online-payment promotion does not. Branch on that, not on the invoice code, or card promotions will land in your reports as phantom bundles. (`online_payment_discount` appears on the cart during checkout; it never becomes an Order total.)

Three conventions to internalize

1. `discount_percentage` on a line is a **fraction** (`0.1` for 10%, with float dust on 4-decimal bases), while `discounts.coupon.percentage` / `discounts.discount_rule.percentage` are **percent** (`10`, `15`) – on the same line.
2. `is_discounted` is `true` for automatic/bundle (`discount_rule`) lines and `false` for coupon lines. It does not mean "has any discount".
3. A coupon's own discount total equals the sum of everything it caused (line cut + free shipping + free COD).

EXAMPLE

One coupon doing three things (observed order, test values). Subtotal 79.99 , coupon freeship (10% + free shipping + free COD): invoice lines sub_totals_before_vat 79.99 , coupon -7.999 , sub_totals_after_coupon_discount 71.991 , taxable_amount 71.991 , vat 10.79865 (after discount), sub_totals_after_vat 82.78965 , shipping 15.00 , free_shipping_coupon -15.00 , zid_cod 5.00 , coupon_cod_discount -5.00 , total 82.78965 .

EXAMPLE

Bundle 15%-off-one (observed order, test values). Lines VAT-inclusive 150.64 + 77.22 (struck from 90.85 with the bundle name on the line): sub_totals_before_vat 209.99 , products_discount -11.85 , sub_totals_after_products_discount 198.14 , taxable_amount 198.14 , vat 29.721 , total 247.861 .

WATCH OUT

Bundle badges render on every product card, but the rule fires only on its configured products — a 3-item mixed cart got no discount while the qualifying pair did. Never infer eligibility from badges; the fired line carries the bundle name with a struck price in the cart and `discount_rule` in the API.

PART 1 • CONCEPTS

1.11 Customer wallets and cashback

KEY TAKEAWAY

Zid is the system of record for wallet funds. Post movements as journal entries, never mirror a balance, and never blind-retry a credit or debit.

Store credit is a real, API-manageable balance on Zid, and it reaches an ERP as two very different things. Keep them apart:

- **Wallet management** — crediting and debiting a customer's balance, reading balances and transactions, and configuring cashback rules — is available through the [Customer Wallets](#) Merchant API.
- **Wallet redemption at checkout** is a cart mechanism. It never appears as a field on the Order. Its effect reaches you only through `payment.invoice[]`, which is exactly why [§1.10](#) insists the invoice breakdown is authoritative.

Headers for this family are `X-Manager-Token` + `Store-Id` (no `Role`) — see [§1.2](#). Scopes: `third_customer_wallet_read` / `third_customer_wallet_write`.

Operation	Path (under https://api.zid.sa/v1/customer-wallets/)	Reference
List wallets	<code>GET customers/</code>	List Customer Wallets
Read one balance	<code>GET customers/{customer_id}/balance/</code>	Get Customer Wallet Balance
Credit	<code>POST customers/{customer_id}/credit/</code>	Credit Customer Wallet
Debit	<code>POST customers/{customer_id}/debit/</code>	Debit Customer Wallet
Bulk credit / debit	<code>POST customers/bulk-credit/</code> , <code>POST customers/bulk-debit/</code>	Bulk Credit , Bulk Debit
Transactions	<code>GET customers/{customer_id}/transactions/</code>	Customer Wallet Transactions
Pending cashback	<code>GET customers/{customer_id}/pending-cashbacks/</code>	Customer Pending Cashbacks
Export	<code>GET customers/export/</code>	Export Customer Wallets
Cashback rules	<code>GET/POST cashback-rules/</code> , <code>GET/PUT/DELETE cashback-rules/{id}/</code>	Cashback Rules
Analytics	<code>GET analytics/</code>	Wallet Analytics

What an ERP integration has to get right

- **Four balance figures, not one.** A balance response carries `total_balance` (including funds on hold), `balance` (available to spend), `held_balance`, `currency`, and `pending_cashback` (earned but not yet credited). Decide which one your GL posts and which one your support screens show — the same distinction as available-vs-on-hand stock in [§1.8](#).
- **Credits can expire.** A credit accepts an optional `expires_at`. Expiring store credit is a liability with a date on it; if your ERP has no concept of expiry, record the date anyway so the write-back is auditable.
- **Credit and debit are not idempotent, and the money moves before the bookkeeping.** `amount` must be greater than zero; the response returns a `transaction_id`. If a call times out or returns `5xx`, read the balance and transactions before retrying — a blind retry credits the customer twice. Store the `transaction_id` against your ERP document.

- **amount** comes back as a string while you write it as a number. Parse to decimal at the boundary, exactly as with refunds in §3.3.
- **Bulk adjustments cap at 100 customers per call** and are applied per customer, so a bulk call can partially succeed. Reconcile from the response, never from your request.
- **Two gates produce a 403 that looks like a scope problem but is not:** the store's plan must include the customer-wallets feature, and the store's currency must be a GCC currency (SAR , AED , QAR , KWD , BHD , OMR). A correctly scoped app on a non-GCC store will be refused. Detect this at install and disable the feature in your UI rather than retrying.
- **Do not mirror balances.** Zid is the system of record for wallet funds. Post movements (credits, debits, redemptions seen on invoices) as journal entries; never run a parallel balance that can drift.
- Older `customers/{id}/wallet/<action>/` paths still answer for backward compatibility but are slated for removal. Build only on the paths above.

PART 1 • CONCEPTS

1.12 Marketplace orders (Mazeed, Trendyol): identify, invoice, return

KEY TAKEAWAY

Marketplace orders look like normal Orders but follow different invoicing, return, and identity rules. Distinguish the three types by markers, never by name similarity.

Misclassifying a marketplace order as a standard storefront sale produces duplicate invoices, rejected partial returns, and campaigns sent to placeholder contacts.

TYPE 1

Mazeed direct

Markers: `payment_method` code starts with `prepaid_mazeed`, or `source = mazeed_marketplace`.

Invoicing: do nothing. Zid issues the B2C consumer e-invoice itself as seller (`MZ...` numbering for invoices, `MZCN...` for notes). Detect via `extra_data.invoice_id` / `invoice_link` and skip invoice creation entirely – the platform never issues a second document over an existing one, and neither should you. Post the sale against Zid's document references, not your own invoice series.

Returns: full-reverse only – a partial return is rejected outright (a full reverse with a partial refund is fine). Reconcile credit notes against the platform's per-reverse map (`mazeed_return_credit_notes[reverseId]`); never self-credit. Cancel-after-return and return-after-cancel are both refused platform-side – treat them as terminal states, not retry queues.

Commission and fee splits are not on these lines at all: they live on a separate merchant commission invoice (`AMINV` series), which is a fee posting, never a customer posting – keep the two series apart in the GL.

TYPE 2

Trendyol revenue-share

Markers: `is_marketplace_order = true` and `prepaid_trendyol` and `reseller_id = mazeed`, paid.

Handle exactly like Mazeed direct: same Zid-as-seller B2C invoice, same skip rules, same full-only returns, same note map. Do not build separate invoicing logic for it – it is the same platform path.

TYPE 3

Trendyol classic and other processing-fee marketplaces

Markers: `is_marketplace_order = true` with no `mazeed` reseller, plus `source = external.*`.

Invoicing: do nothing on the ZATCA side. No Zid-side consumer invoice exists for these – `merchant_invoicing_*` fields will be absent (expect that; do not flag it as missing data) and the merchant EGS flow explicitly excludes them. The compliant invoice lives marketplace-side. Record the sale referencing the marketplace order id. What Zid does generate is a bilingual PDF for the marketplace (name-unmasked), which is a document handoff, not a tax posting.

WATCH OUT

Do not identify marketplace orders by `marketplace_order_id` or `marketplace_package_id`. Those fields live inside a `marketplace_meta` block that the platform delivers only to first-party clients — your app never receives it, on any payload type. (They are still accepted as List filters, so you can search by an id you already hold; you just cannot read one back.) Identify these Orders from `is_marketplace_order`, `source_code`, `payment_method`, and `reseller_meta`, and keep your own reference to the marketplace id from whatever system supplied it.

Identity (all marketplace types): PII is masked by design. Customer name, email, and phone arrive as guest/placeholder values; the shipping city reads `Other` with the real city only in payload fields and the postcode only in address meta. Never send campaigns to masked contacts, never treat them as deliverable, never "complete" them by guessing. Pull live PII only through audited surfaces for a genuine fulfillment need — name-only for invoicing and printing — and store the minimum. Webhook and third-party API views stay masked except narrow shipping-app cases; the merchant dashboard sees more than your app ever will, and that asymmetry is intentional.

Clearance vs reporting is determined outside the Merchant API surface (per document, inside the invoicing path) — all Mazeed-path documents in this program are B2C; do not assert phase behavior from Order fields.

02

PART 2

INTERFACE DESIGN DECISIONS

2.1 Decisions to settle per data object

2.2 Mapping Zid concepts onto any ERP

PART 2 • INTERFACE DESIGN

2.1 Decisions to settle per data object

KEY TAKEAWAY

Answer these per data object, not for the integration as a whole. One-way sync is dramatically simpler and correct more often than teams expect.

- 1. Direction** – inbound, outbound, both, or out of scope for v1? Each two-way object roughly triples design work.
- 2. Source of truth, per field** – a product's price is usually ERP-owned while its marketing description is Zid-owned. Exactly one owner per field; where both sides must edit, write the conflict rule down now.

3. The nine questions that remove most later ambiguity:

1

Where is stock edited after go-live?
(If "both", settle the process first.)

2

Volumes – SKUs, peak Orders/day, customers?

3

More than one legal entity – which receives Zid Orders?

4

Acceptable lag – seconds, minutes, hourly, overnight?

5

Where does the Zid Order number live in the ERP?

6

Guest checkout – one shared web customer or a record per Order?

7

How do warehouses map to Inventory Locations?

8

Are prices VAT-inclusive, and does the ERP match?

9

Who owns this integration in twelve months, by name?

- 4. Cross-reference table** (your own store, backed up): `object_type | zid_id | erp_id | last_synced_at | content_hash | status`. Never match on SKU, email, or phone instead – all three change. Store a content hash to skip no-op writes (60 req/min punishes blind rewrites). Key by store ID even for a private app; keep both the numeric store ID and the store UUID. Losing this table duplicates everything on the next sync.
- 5. Reconciliation job** (scheduled, reporting even on clean days): line arithmetic per Order, line-sum-to-header, invoice `total` match, VAT-after-discount match, currency-intent check, Credit Note totals with `original_invoice_number` linkage, gateway settlement net of fees to a tolerance. This catches what never errors: missed webhook deliveries during an outage are never replayed, so reconciliation is the only backstop.

PART 2 • INTERFACE DESIGN

2.2 Mapping Zid concepts onto any ERP

KEY TAKEAWAY

Names differ per system; the shapes do not. Use this as the starting point for a field-mapping document, and resolve every "watch out" before the first load.

NOTE

If your ERP has no equivalent for a concept, record the Zid value in a custom field anyway. Every field dropped at integration time becomes an unanswerable support question later.

Zid concept	Typical ERP counterpart	Watch out
Order	Sales Order / Sales Invoice	One Zid Order can become two ERP documents (order + invoice). Decide which carries the Zid <code>id</code> and which carries <code>invoice_number</code>
Order Status	Document status / workflow state	Map ERP onto Zid, not the reverse (§1.7)
<code>payment.invoice[]</code>	Invoice totals block	Authoritative. Never recompute from mechanisms (§1.10)
Reverse Order	Return order / RMA	Creating one confirms a Credit Note immediately – no draft stage (§3.3)
Credit Note	Credit memo	Arrives already confirmed; link via <code>original_invoice_number</code>
Refund	Payment out / bank entry	Gateway-handled and merchant-managed refunds post differently
Product (parent)	– nothing –	Never map a parent to an item; map its variants
Product Variant	Item / SKU / article	Carries its own price, cost, weight and stock
Custom Option / Input Field	Line-level configuration or surcharge	Contributes additions to the line value – ignoring it understates every configured line
Product Stock	Stock on hand per warehouse	Decide available-vs-on-hand once (§1.8)
Inventory Location	Warehouse / storage location	One stock record per product per location
Customer	Business partner / debtor / AR account	Identity key is phone, not email (§1.9)
Business customer fields	Tax registration on the partner record	<code>tax_number</code> + <code>commercial_registration</code> are what make a B2B invoice claimable
Coupon / Bundle Offer	Discount / promotion	Reporting inputs only; post the invoice lines
Customer Wallet	Store credit / customer deposit liability	Zid is the system of record; post movements, never mirror the balance (§1.11)
Loyalty points	Loyalty liability	Read-only for reconciliation; never synchronize balances
COD fee, shipping, VAT lines	Separate GL accounts	Each has its own invoice code; do not fold them into the goods line

03

PART 3

BUILDING IT

3.1 Products and stock sync

3.2 Orders: list, view, create

3.2a Order field reference

3.3 Returns, Credit Notes, refunds

3.4 Coupons and Bundle Offers

3.5 Webhook receiver

3.6 Errors, limits, silent incompleteness

3.7 Money, VAT, invoicing, and reconciliation

3.8 Testing, cutover, go-live

PART 3 • BUILDING IT

3.1 Products and stock sync

KEY TAKEAWAY

Initial load and delta sync are separate code paths. The load must be paced against the limits, checkpointed, and resumable.

State the arithmetic to the business first: 5,000 products at one request each is ~83 minutes at the limit, before retries.

- Page with `page_size` up to 100, follow `next` links, order by `-updated_at` for deltas, deduplicate by UUID.
- Write stock with [Bulk Update Product Stock](#); store a content hash and skip unchanged records.
- For the very first load, consider [Import Products via CSV/XLSX](#) instead of thousands of API writes.
- Reconcile from write responses, never by merging your request into cached data.
- Product classes, variants, options, and the write/delete semantics in [§1.8](#) are the complete contract – reread it before coding the mapper.

PART 3 • BUILDING IT

3.2 Orders: list, view, create

KEY TAKEAWAY

List Orders omits line items by default – pass `payload_type=default` whenever you need lines or money. Validate before you create, and never resend a create on a timeout.

List orders omits line items by default. The default `payload_type=simple` returns Orders with no product lines and no money detail – every sync that needs lines or money must pass `payload_type=default`. `payment.invoice[]` exists only on the full payload.

Listing

- **Pagination, two modes. Page-number:** one-based `page` (default 1), `per_page` default 50, **max 500**; no page links (use `total_order_count`); default sort `id` descending; an out-of-range page returns `200` with an empty array (a pagination bug looks like "no more data"). **Cursor:** pass `cursor` and the endpoint switches to cursor mode, which does not shift as Orders change – prefer it for any walk of more than a page or two. Page-number pagination re-shuffles under you as new Orders arrive, so for delta sync use `created_at` / `updated_at` windows with a watermark, and **filter on `updated_at`** (an Order that changes status days later keeps its creation date).
- `date_from` / `date_to` apply to `date_attribute` (`created_at` / `updated_at`); date-only values use Asia/Riyadh day boundaries.
- **Filters that matter:** `source` (API-created Orders are recorded as API), `search_term`, `sku`, `order_id`, `status` / `order_status` / `status_ids`, `inventory_location_id`, `payment_status`, `payment_method`, `transaction_reference`, `shipping_method` / `shipping_method_code`, `shipping_city`, `shipping_country`, `tags`, `customer_id`, `customer_group`, `customer_phone` + `country_code`, `integrated_loyalty`, marketplace IDs, gift/preorder flags. Values inside one array match any; filters combine with AND. `marketplace_order_id` is accepted as a filter but is not returned to apps – see §1.12.
- **Payload comparison (same Order):** **simple** carries identity, status, customer, `payment.method` only, `gift_card_details`, and `point_type` – no lines, no coupon, no invoice. **Default** adds `products`, `coupon` + `coupons[]`, `cashback`, and the full `payment` block including `invoice[]`. **View** adds `histories`, `refunds`, `payments[]`, `payment_summary`, `can_request_order_due_payment`, and invoice settings/link/number. Field by field, see §3.2a. Three further payload types exist (`tiny`, `print`, `print_summary`) that serve dashboard and document rendering – an ERP integration should not build on them.
- **inventory_allocations** is `[]` on every list payload. For which location fulfills which line, read [Order Fulfillment Inventories](#) (`GET /v1/managers/store/orders/{order}/fulfillment-inventories`, scope `third_order_read`) – the endpoint a WMS mapping needs.
- `source` is the localized display name, `source_code` the raw key – filter and branch on `source_code`.
- **Multi-currency:** `order_total` (Order currency) vs `transaction_amount` (transacted), with `has_different_transaction_currency` flagging a narrow case (online-card payment in non-store currency). The backend computes both figures – decide once which posts to the GL in which currency.
- For dashboards and reconciliation summaries, [Get Orders Highlights](#) returns aggregates in one request instead of paging.

Creating Orders from the ERP

Two operations share one body: [Validate Order Request](#) (creates nothing) and [Create Order](#) (creates the Order). **Always validate first** – create is not idempotent and takes no idempotency key.

- **Products are selected by SKU**, not product ID – keep SKUs current or resolve ID → SKU before posting.

- `shipping_method` is required only when the cart requires an address; `payment_method` is `{id}` or `{type: prepaid, code}`; payment-link options need a future expiry; `full_name` caps at 32 characters; the Saudi national short address **KSA ONLY** follows `^[A-Za-z]{4}[0-9]{4}$`.
- Creation rechecks stock, reuses customers by phone, applies coupons, allocates stock, and records source as API. **On timeout or 5xx : do not resend** — search for the Order by source, customer, and creation window first.
- Order-to-payment is not one-to-one (partial and later settlement exist). Payment status cannot distinguish refund from non-payment — read the Reverse Order and Credit Note instead.

Tags and comments

Order tags (3–128 chars, max ten) are find-or-create and idempotent — safe to re-assert. **Comments** (1–255 chars) append a distinct record every call with no idempotency — a retry duplicates. Neither emits a webhook you can rely on (`order.tagged` fires only from POS). Empty history means the merchant lacks that capability, not that nothing happened.

PART 3 • BUILDING IT

3.2a Order field reference

KEY TAKEAWAY

The field inventory an ERP mapper needs, with where each field appears. **S** = `payload_type=simple` (the List default), **D** = `payload_type=default` (List), **V** = View Order only. Fields marked **S•D•V** are everywhere.

WATCH OUT

Order webhooks carry the full detail shape, so "V only" does not mean "not in the webhook". An `order.*` delivery includes `histories`, `refunds`, `payments[]`, `payment_summary` and `can_request_order_due_payment` – everything marked V only below – because the payload is built with the detail serializer. If you are reacting to a webhook you usually already hold what you would otherwise call [View Order](#) for; spend the request only when you need something the event genuinely lacks.

Two conventions apply throughout and are not repeated per row:

- **string twins.** Most money fields have a formatted sibling (`order_total` / `order_total_string`, `net_price` / `net_price_string`). Compute on the bare field, display the `_string`. Some line tax fields add a `_per_item` variant too.
- **code vs display name.** Where a field pairs a code with a localized label (`order_status.code` / `order_status.name`, `source_code` / `source`, `payment.method.code` / `.name`), branch only on the code.

WATCH OUT

`store_uuid`, `store_logo` and `product_ids` exist in the serializer but are gated behind opt-in flags that the Merchant API does not pass – **they never appear on a Merchant API order**. Get the store UUID from your install record, not from an order.

Identity, status, dates

Field	Availability	ERP note
<code>id</code>	S•D•V	Zid's numeric order id. Your primary cross-reference key
<code>code</code>	S•D•V	Human-readable order number – what the merchant quotes you
<code>invoice_number</code>	S•D•V	Falls back to <code>id</code> when no invoice exists, so it is not proof an invoice was issued (§3.7)
<code>order_status</code>	S•D•V	Object: <code>{code, name}</code> . Map on <code>code</code> (§1.7)
<code>display_status</code>	S•D•V	An object <code>{id, code, color, name}</code> – the merchant-facing status, which is where a custom sub-status surfaces. Mirrors <code>order_status</code> on stores without custom statuses. Branch on <code>order_status.code</code> for lifecycle logic; read this one to show the merchant what they see
<code>source</code> / <code>source_code</code>	S•D•V	Localized name / raw key. Filter and branch on <code>source_code</code>
<code>created_at</code> / <code>updated_at</code>	S•D•V	UTC. Delta sync on <code>updated_at</code>
<code>issue_date</code>	S•D•V	Not a timestamp – a localized display string (<code>"05-10-2026 11:20 ص"</code>), store timezone, Arabic. Never parse it (§1.6)
<code>delivered_at</code>	D•V	Third distinct date. Revenue recognition usually keys off this one

Field	Availability	ERP note
<code>store_id</code>	S-D-V	Numeric store id; the tenant key for a multi-store integration
<code>store_name</code> , <code>store_url</code> , <code>order_url</code>	S-D-V	<code>order_url</code> is the merchant-facing invoice page
<code>import_id</code>	S-D-V	Set when the order was imported, not placed. Useful to exclude from sales reporting
<code>language</code>	D-V	Buyer's language – pick the right side of every <code>{ar, en}</code> map with it
<code>tags</code>	S-D-V	Order tags (§3.2). ZATCA posting status arrives here (§3.7)
<code>edits_count</code>	D-V	Non-zero means totals were edited after placement – reconcile, don't assume
<code>is_reactivated</code>	D-V	Order was revived after cancellation

Money, currency, payment

Field	Availability	ERP note
<code>order_total</code>	S-D-V	Order-currency total. The headline figure
<code>currency_code</code>	S-D-V	Order currency. Required on every GL posting
<code>currency</code>	D-V	Two sub-objects: <code>order_currency</code> and <code>order_store_currency</code> , each <code>{id, code, exchange_rate}</code> ; <code>exchange_rate</code> can be <code>null</code> when the currencies match (§1.6a)
<code>transaction_amount</code>	S-D-V	Amount actually transacted, in store currency
<code>has_different_transaction_currency</code>	S-D-V	ZidPay-only flag – not a general multi-currency test (§1.6a)
<code>transaction_reference</code>	S-D-V	Gateway reference. This is your bank/settlement reconciliation key – and it is also a List filter
<code>payment_status</code>	S-D-V	Cannot distinguish refunded from unpaid. Read the Reverse Order and Credit Note instead
<code>payment.method</code>	S-D-V	<code>{name, code, type}</code> ; on D-V adds bank-transfer detail (<code>transaction_bank</code> , <code>transaction_slip</code> , <code>transaction_sender_name</code>) and transaction status
<code>payment.invoice[]</code>	D-V	The authoritative totals breakdown. Post this, never a recomputation (§1.10)
<code>payment.split_payments</code>	POS orders only	Split tender at the till
<code>payments[]</code>	V only	Individual payment records – this is what makes order-to-payment one-to-many
<code>payment_summary</code>	V only	Paid / remaining rollup. Use it for partial-settlement logic
<code>can_request_order_due_payment</code>	V only	Signals an outstanding balance
<code>payment_link</code>	D-V	Payment-link URL when the order was placed that way
<code>payment_network</code>	D-V	Card network, where the gateway reports one
<code>invoice_link</code>	D-V	Zid-hosted invoice document
<code>service_fee_invoice_link</code>	D-V	Separate fee document – a fee posting, not a customer posting

Field	Availability	ERP note
<code>return_invoice_number</code>	D-V (POS)	Return document number
<code>cashback</code> , <code>nd_cashback</code>	D-V	Reporting fields (§1.10)
<code>coupon</code> , <code>coupons[]</code>	D-V	Always read the array
<code>marketplace_commission</code>	D-V	Commission on marketplace orders (§1.12)
<code>refunds[]</code>	V only	Refund records against the order

Parties

Three different parties can appear on one order; an ERP that collapses them will print the wrong name on a document.

Block	Availability	Fields
<code>customer</code>	S-D-V	<code>id</code> , <code>name</code> , <code>email</code> , <code>mobile</code> , <code>note</code> , <code>verified</code> , <code>type</code> , <code>business_name</code> , <code>tax_number</code> , <code>commercial_registration</code> , <code>orders_count</code> , <code>order_total_payments</code> . The B2B trio is what makes a tax invoice claimable (§1.9)
<code>shipping.address</code>	S-D-V	<code>formatted_address</code> , <code>street</code> , <code>district</code> , <code>city</code> , <code>country</code> , <code>lat</code> , <code>lng</code> , <code>short_address</code> , <code>meta</code> . Unresolved cities are omitted, not null (§3.6)
<code>consignee</code>	D-V	<code>name</code> , <code>email</code> , <code>mobile</code> , plus <code>requires_recipient_address</code> and <code>address_collection_link</code> . Present when <code>has_different_consignee</code> is true – the recipient is not the buyer, which changes who the delivery document names
<code>is_guest_customer</code>	S-D-V	Answers the guest-checkout design question in Part 2 directly
<code>inventory_address</code>	D-V	Pickup location or Inventory Location the order ships from

Fulfillment, flags, and the rest

Field	Availability	ERP note
<code>requires_shipping</code>	S-D-V	False for all-digital orders – no shipping line, and <code>indelivery</code> is rejected
<code>shipping.method</code>	S-D-V	Chosen method; <code>shipping_method_code</code> is the raw key
<code>should_merchant_set_shipping_method</code>	S	Merchant still has to pick a method
<code>expected_shipping_method_type</code>	D-V	Intended type before selection
<code>weight</code> , <code>weight_cost_details</code>	D-V	Billable weight and how the cost was derived
<code>packages_count</code>	D-V	Defaults to 1
<code>cod_confirmed</code>	S-D-V	COD confirmation state
<code>reverse_order_label_requests[]</code>	S-D-V	Return labels on the order itself – a cheap trigger for the returns flow (§3.3). A singular <code>reverse_order_label_request</code> (latest, or <code>null</code>) sits alongside it
<code>return_policy</code>	D-V	Localized policy text at time of order

Field	Availability	ERP note
<code>histories[]</code>	V only	Status history. Empty means the merchant lacks the capability
<code>products[]</code> , <code>products_count</code>	D•V	Lines. Absent on simple
<code>is_potential_fraud</code>	S•D•V	The fraud flag
<code>is_gift_order</code> , <code>gift_card_details</code> , <code>gift_message</code>	S•D•V (message D•V)	<code>gift_card_details</code> is the one money-ish block present even on simple
<code>is_preorder</code> , <code>is_preorder_allocated</code> , <code>preorder_campaign</code>	D•V	Pre-orders allocate stock later – do not post as shippable
<code>is_on_demand</code>	S•D•V	Made-to-order
<code>is_quick_checkout_order</code>	S•D•V	Quick-checkout cart
<code>is_referral_order</code>	D•V	Came via referral
<code>is_marketplace_order</code> , <code>reseller_meta</code> , <code>is_reseller_transaction</code>	S•D•V (<code>reseller_meta</code> D•V)	The reliable marketplace markers (§1.12)
<code>pos_transaction_data</code> , <code>pos_invoice_number</code> , <code>pos_payment_method_editable</code>	D•V (POS)	POS-originated orders
<code>zidship_ticket_number</code>	D•V	Shipping support reference
<code>customer_note</code>	D•V	Buyer's note – often carries delivery instructions a WMS needs
<code>digital_products_access_token</code>	D•V	Never log or forward. Grants download access
<code>marketing_source</code>	S•D•V	Attribution block: full UTM tuple, <code>referrer</code> , <code>referrer_code</code> , <code>platform</code> , <code>ip</code> , <code>ip_country</code> , <code>click_ids</code> , <code>visitor_id</code> . <code>null</code> when no referrer row exists
<code>marketplace_meta</code>	first-party only	Not delivered to partner apps – see the warning in §1.12
<code>point_type</code>	S	Integrated loyalty point type

Order line fields

`products[]`, default and view only. The line money model is the densest part of the payload and the most common source of ERP drift.

Group	Fields	What to post
Identity	<code>id</code> , <code>order_product_id</code> , <code>sku</code> , <code>barcode</code> , <code>name</code> (<code>{ar, en}</code>), <code>short_description</code> , <code>product_class</code> , <code>unit</code> , <code>weight</code> , <code>images</code> , <code>review_url</code>	<code>order_product_id</code> is the key partial returns are quoted against (§3.3). <code>product_class</code> , <code>barcode</code> and <code>weight</code> are nullable; line <code>weight</code> is <code>{value, unit}</code> or <code>null</code>
Parent linkage	<code>parent_id</code> , <code>parent_name</code> , <code>parent_sku</code>	Present on variant lines. Map the line, not the parent (§1.8)

Group	Fields	What to post
Base price	<code>price</code> , <code>price_before</code> , <code>net_price</code> , <code>gross_price</code> , <code>net_sale_price</code> , <code>gross_sale_price</code>	Net is pre-VAT, gross is VAT-inclusive. Pick one convention per your VAT setting and never mix
Additions	<code>additions_price</code> , <code>net_additions_price</code> , <code>gross_additions_price</code>	Custom options and input fields (§1.8)
Price including additions	<code>price_with_additions</code> , <code>net_price_with_additions</code>	This is the unit price to post. It resolves sale-vs-list as well as additions: on an observed line, <code>net_price</code> was 99.99 (list) while <code>net_price_with_additions</code> was 79.99 (the sale price actually charged). Posting bare <code>net_price</code> there overstates the line by 20.00
Quantity & totals	<code>quantity</code> , <code>total</code> , <code>total_before</code> , <code>discounted_total</code>	<code>total_before</code> is pre-discount; <code>discounted_total</code> is post
Discount	<code>discount_percentage</code> (fraction), <code>discounts</code> ([] when empty, object when populated), <code>is_discounted</code>	Read the three conventions in §1.10 before mapping, and the JSON-type note in §1.6
Tax	<code>tax_percentage</code> , <code>tax_amount</code> , <code>discounted_tax_amount</code> , <code>is_taxable</code>	VAT after discount. <code>discounted_tax_amount</code> is the figure that ties to the invoice
Configuration	<code>options</code> , <code>custom_fields</code> , <code>vouchers</code> , <code>downloadables</code> , <code>downloadables_zipped</code>	Carry these onto the ERP line or the document loses what the buyer actually ordered
Fulfillment	<code>inventory_allocations</code>	Per-line location. Empty on list payloads – use Order Fulfillment Inventories
Meta	<code>meta</code> , <code>is_external_product</code>	<code>meta.bundle_offer</code> is the bundle identifier (§1.10)

Invoice line shape (`payment.invoice[]`): each entry is `{code, title, value, value_string}`. Branch on `code`, compute on `value` (§1.6).

PART 3 • BUILDING IT

3.3 Returns, Credit Notes, refunds

KEY TAKEAWAY

Creating a Reverse Order creates and confirms a Credit Note atomically — no draft, no approval step. There is no reverse-order webhook; trigger off `reverse_in_progress` and `order.refund.create`.

This is the only Zid flow that creates a financial document automatically. For ERPs that review credit memos before posting, surface this constraint early.

EXAMPLE

Observed behavior worth knowing. On unpaid (e.g. COD-pending) Orders, a full return + refund and a cancellation both completed correctly with no visible Credit Note — notes follow paid money, not merchandise movement. Refund `amount` reads back as a 2-decimal string ("138.00") while writes are numeric — parse to decimal at the boundary.

The seven steps, in order

- 1 **View order for return** — `GET /store/orders/{order}/view-for-return`. Lines and totals show remaining returnable quantities (prior returns subtracted). Refresh before every additional partial return. Empty `products` with zero totals is valid (nothing left), not an error.
- 2 **Calculate reverse totals** — `POST /reverse-credit-note/totals` (note: no `/store` prefix). Read-like and safe to retry, but requires write scope. Returns exactly five lines in fixed order: `sub_totals` (net × qty), `discounts` (unit discount × qty), `sub_total_after_discounts`, `vat` (unit tax after discounts × qty), `total`. `order_id` required; partial mode needs quantities keyed by `order_product_id`.
- 3 **Create the Reverse Order** — `POST /store/reverse-orders`. Precondition: `delivered` or `partially_reversed`. Required: consignee name/mobile/city/address ×2, `order_id`, `reason` as a JSON object `{id, name}` (a bare string fails), and exactly one of `inventory_address_id` / `inventory_location_id`. In full mode the server recomputes partiality — your flag is not authoritative. No idempotency key: on timeout, View before retrying or you will duplicate the Credit Note. `Reverse reasons` list global + store-specific entries, and `adding one` is find-or-create by name.
- 4 **Waybill (optional)** — `POST /store/reverse-orders/waybill`. One per Reverse Order; needs eligible Zid shipping. Provider work can fail after validation — fetch before retrying.
- 5 **Classify returned products** — `PUT /store/reverse-orders/products`. Precondition: `reverse_in_progress`. Atomic across lines: `received + not_received + damaged = reversed_quantity` (strict equality). `received` restocks and feeds `refund_total`; the other two are debit adjustments without restock. Posting all returned units as restocked is the most likely modeling error. Result: `reversed` (all quantities returned) or `partially_reversed`.
- 6 **Create the refund** — `POST /store/reverse-orders/refund` (`multipart/form-data`): `order_reverse_id` (UUID), `amount ≥ 0.01`, `payment_method`, `notes ≤ 1000`. Exactly one refund per Reverse Order (a second attempt is rejected); the cap is the remaining refundable amount, not the original total. Gateway methods (`zidpay`, `payment_link.zidpay`, `tabby`, `tamara`) invoke provider handling and settle via payout; the rest record merchant-managed bookkeeping — post them differently. The offered methods list is narrower than the full enum — offer from `available_refund_payment_methods`. Classify first so `refund_total` is known. Provider work can precede persistence: reconcile before retrying, never blind-retry. `Receipt upload` (image, ≤ 5 MB, ≤ 4096×4096) applies to bank-transfer refunds and replaces any previous receipt; it is optional at creation.
- 7 **Read Credit Notes** — `GET /store/orders/{orderId}/credit-notes`. Only confirmed notes appear; notes also arise from cancellations and edits. Monetary fields carry numeric + formatted forms — compute on numeric. Filter item lines on `type` (`product`, `shipping_method`, `payment_method`, `coupon`), not on the `shipping` boolean (it does not mark the shipping line). `original_invoice_number` links to the source document; store the ZATCA objects and pass them through. More than ten reads for one Order in five minutes returns `429` — trigger from status changes, never tight-poll.

PART 3 • BUILDING IT

3.4 Coupons and Bundle Offers

KEY TAKEAWAY

Coupons and Bundle Offers are fully writable, but they are reporting inputs: the money your ERP posts comes from `payment.invoice[]`.

- **Coupons:** **Create Coupon** validation — name 3–128; code ≤ 20 chars, no whitespace; `discount_type` fixed / percentage / no-discount; dates `YYYY-MM-DD` with start ≤ end; **condition operators are only `in`, `=`, `type`**; **progressive strategy requires `applying_method=AUTOMATIC`, `apply_to=all`, and non-overlapping active date ranges** — only one overlapping progressive coupon per store; tier values are decimal strings. A coupon can combine a line cut, free shipping, and free COD in one object (its discount total is the sum of all three). **Bulk generation** exists for campaign volumes.
- **Bundle Offers support create/update/delete** with write scope (they are not read-only). Badges render on all product cards, but firing is product- and condition-specific — a mixed cart may get nothing while the qualifying pair gets the cut. The fired line carries the bundle name with a struck price in the cart and `discount_rule` in the API. Never promise eligibility from badges. To identify bundle Orders in your sync, see the rule in **§1.10**: associated offer from `products[].meta.bundle_offer.id`, fired money from the `discounts.discount_rule` + `is_discounted` + `products_discount` triple.
- **Progressive (tiered) coupons self-apply** once the subtotal crosses a tier and advertise with an on-cart progress meter — no code is typed, and they still appear in `coupon` / `coupons[]` on the Order.

PART 3 • BUILDING IT

3.5 Webhook receiver

KEY TAKEAWAY

Verify the signature over the raw body bytes, record the receipt durably, acknowledge with 200/201/202/204, then process. Idempotency is yours end to end, and missed events are never replayed.

Subscribe with [Create Webhook](#) (`event` , `HTTPS target_url` , `original_id` , optional `conditions` , optional Basic `username` + `password`). [List](#), [Health Summary](#), [Broken Webhooks](#), [Recover](#), and [per-subscriber delete](#) endpoints exist – recovery replaces registrations without replaying missed events, so reconciliation covers outages. Monitoring guidance: [Webhook Health Tracking](#).

Request contract, exactly as observed live

- **Route and allowlist on** `Webhook-Event` (exact name); `Webhook-Resource` (name minus last segment) is metadata only.
- **With both credentials configured** you receive `Authorization: Basic <base64(user:pass)>` and `X-Signature: <lowercase hex HMAC-SHA256 over the exact raw body bytes, key = password>`. Verify over the received bytes before parsing JSON, in constant time. Re-serializing the parsed object (with any encoder) changes escaping and fails verification. Without both credentials, neither header is sent, and their absence proves nothing.
- **Acknowledge with exactly 200 , 201 , 202 , or 204** – after durably recording the receipt, not after downstream work. Anything else (including 206 , redirects, and 429) counts as failed delivery. Load-shedding with 429 loses events silently.
- **Delivery:** 10-second request timeout, up to three attempts with backoff, no ordering guarantee, no exactly-once, no published schedule SLA. Broken subscriptions stop new and retried deliveries. Test deliveries carry `X-Webhook-Test: true` (10 per 60 seconds) and prove reachability only.
- **Own-write suppression:** for most resource events Zid does not deliver back to the app whose Merchant API write caused them (abandoned carts excepted) – use the API response as your write result, never wait for a webhook as acknowledgement.

Verifying the signature

Capture the raw body before any JSON middleware touches it.

```
// Node / Express – note the raw body parser, not express.json()
import crypto from "node:crypto";

app.post("/zid/webhook",
  express.raw({ type: "*/*" }), // req.body stays a Buffer
  (req, res) => {
    const expected = crypto
      .createHmac("sha256", process.env.ZID_WEBHOOK_PASSWORD)
      .update(req.body) // raw bytes, never JSON.stringify
      .digest("hex");
    const received = String(req.get("X-Signature") || "");
    const ok = expected.length === received.length &&
      crypto.timingSafeEqual(Buffer.from(expected), Buffer.from(received));
    if (!ok) return res.sendStatus(401);

    recordReceipt(req.get("Webhook-Event"), req.body); // durable write first
    res.sendStatus(200); // then acknowledge
    enqueueProcessing(); // downstream work after
  });
```

NODE.JS

```
<?php // PHP
$raw = file_get_contents('php://input');
$expected = hash_hmac('sha256', $raw, getenv('ZID_WEBHOOK_PASSWORD')); // lowercase hex
$received = $_SERVER['HTTP_X_SIGNATURE'] ?? '';
if (!hash_equals($expected, $received)) { http_response_code(401); exit; }

record_receipt($_SERVER['HTTP_WEBHOOK_EVENT'] ?? '', $raw);
http_response_code(200);
```

PHP

```
# Python / Flask
import hashlib, hmac, os
from flask import request, abort

@app.post("/zid/webhook")
def zid_webhook():
    raw = request.get_data() # bytes, before request.json
    expected = hmac.new(os.environ["ZID_WEBHOOK_PASSWORD"].encode(),
                        raw, hashlib.sha256).hexdigest()
    if not hmac.compare_digest(expected, request.headers.get("X-Signature", "")):
        abort(401)
    record_receipt(request.headers.get("Webhook-Event"), raw)
    return "", 200
```

PYTHON

Idempotency is yours, end to end

No event carries a platform delivery ID and no write takes an idempotency key (tag assignment and reason creation are the find-or-create exceptions). Reserve your own key with the durable receipt, make business writes conditional on current state, read before irreversible actions (refunds, wallet adjustments, postings, stock moves), and define per operation how to find a possibly-created record before resending.

For objects without events, poll narrow `updated_at` windows with slight overlap (idempotency absorbs repeats) behind a watermark – gapless beats exactly-once.

PART 3 • BUILDING IT

3.6 Errors, limits, silent incompleteness

KEY TAKEAWAY

Branch on HTTP status plus the `code` slug, never on localized text. And assume some failures return `200` – build completeness checks into reconciliation.

Limits

60 requests/minute per app per store (leaky bucket), plus per-endpoint limits (Order Credit Notes: 10 reads per Order per 5 minutes). Honor `Retry-After`; back off exponentially with jitter; drain post-outage backlogs at a controlled rate. Merchants with genuine need can request an exception from Zid.

Errors

Response shapes differ by family (see [Responses](#)) – a `status` / `message` object, a `code` / `message` / `payload` object, or the Products-family pair: `{"detail": "<localized message>"}` for auth, permission, not-found and throttle errors, and a field-keyed object of message arrays for validation errors (`{"price": ["..."], "variants.0.sku": ["..."]}`), with nested paths joined by `.` and anything not tied to a field under `non_field_errors`). Code that reads only `detail` will log an empty reason for every `400` in that family. Branch on **HTTP status plus the `code` slug** – never on `description` (localized), and never on the `429` body, whose wording is not consistent across families.

Status	What it means and what to do
401	Refresh once, then reauthorize
403	Scope, manager permission, plan feature, or market eligibility; do not retry (§1.5)
404	Check store context first
400	State-rule violation; read current state
409	Phone duplicate
422	Surface the field; never retry unchanged
5xx on a write	Unknown outcome; search before resending

Silent incompleteness (all `200`, no error)

Default list payloads without lines; empty history without capability; out-of-range pages as empty arrays; unresolved-city addresses omitted; bulk-create responses with only new rows; import `job_id` as acceptance, not completion; `inventory_allocations` empty on list payloads.

NOTE

Build completeness assertions into reconciliation and report on clean days – a monitor silent when healthy is indistinguishable from a dead one.

PART 3 • BUILDING IT

3.7 Money, VAT, invoicing, and reconciliation

KEY TAKEAWAY

Payment status — never Order Status — gates a tax document. Detect per channel whether Zid or your ERP issues the e-invoice, and never issue twice.

B2B buyers carry `type`, `business_name`, `tax_number`, `commercial_registration` for compliant invoices; absent means consumer.

When Zid generates an invoice, and which settings decide it

Two different documents are easy to conflate, and they are gated differently. Settle both before you write invoicing logic.

	PDF document (<code>invoice_link</code>)	E-invoice (ZATCA) KSA ONLY
What it is	A rendered PDF — either an order confirmation or a tax invoice , see below	The cleared/reported tax document, Saudi Arabia only
Gate 1	Store setting <code>is_zid_invoice_generation_enabled</code> (default true); <code>null</code> when off	<code>payment_status</code> is <code>paid</code>
Gate 2	The file must already exist — <code>null</code> until it is rendered	The Order's sales channel has an enabled EGS unit
Payment status	Selects which document you get; it does not gate the field	Gates creation entirely
Also	The dashboard's invoice print action is additionally gated on paid, and excludes imported Orders (<code>import_id</code>)	Channel must map to an EGS unit at all — see below

WATCH OUT

A populated `invoice_link` is not a tax invoice. While `payment_status` is pending, Zid renders an **order confirmation** PDF and `invoice_link` points at it; once the Order is paid the link resolves to the tax invoice variant instead. Observed on a live COD Order: `payment_status: "pending"` with `invoice_link` ending `order_confirmation_<code>.pdf`. Check `payment_status` before filing that URL as a tax document — an ERP that attaches the confirmation to the GL has booked the wrong artefact.

What gates a tax document is payment status, never order status. An Order sitting at `delivered` with `payment_status` still unpaid has no tax invoice and no e-invoice; an Order at `new` that is already paid has both. This is the single most useful thing to know here: **do not trigger invoice logic off Order Status**. COD Orders typically invoice late, when payment is confirmed — which is also why §3.3 observes no Credit Note on an unpaid return.

It can arrive later than the Order. Invoice creation is attempted when the Order is created and again when a payment status change comes through, so an Order that was unpaid at creation gets its invoice at the moment it is marked paid. Expect the invoice fields to be empty on first read and to populate afterwards — poll the Order, or key off `order.payment_status.update` (§1.1). Creation is guarded by a lock, an idempotency key and a per-Order tracker, so Zid will not issue two documents for one Order; your side still must not assume the first read is final.

Which channels get an e-invoice KSA ONLY. The Order's `source_code` must map to an EGS unit family, and that unit must be enabled:

source_code	EGS unit family	Notes
catalog, md, mobile_app, api	store	One unit for the whole family. api is included – Orders your ERP creates do get e-invoiced
pos	pos	Per inventory location : each POS location has its own unit, and the location must be in Saudi Arabia
anything else (marketplace, external.*, reseller paths)	– none –	No Zid-side e-invoice exists. This is the mechanism behind \$1.12

So one merchant can legitimately be e-invoicing from the storefront but not from POS, or from one POS branch and not another. Decide per channel, never per merchant.

What the integration should read. `invoice_settings` – `{is_order_notifications_enabled, is_zid_invoice_generation_enabled}` – is returned by **View Order only**; it is not on any List payload. The same `is_zid_invoice_generation_enabled` flag also appears on the store profile (**Retrieve Store Manager Profile**), which is the cheaper place to read it once at startup. If it is false the merchant has turned Zid's invoice rendering off, and your integration is the only thing producing a document – which makes the native-vs-ERP-side decision below not a choice but a given.

WATCH OUT

`invoice_number` falls back to the Order `id` when no invoice exists, so a populated `invoice_number` is **not** evidence that an invoice was issued. Test `payment_status`, and `invoice_link` for the document itself.

Who issues the e-invoice: native vs ERP-side KSA ONLY

Saudi Arabia only. Everything in this subsection is ZATCA e-invoicing and applies to Saudi stores; for a store in another market, skip to the VAT-settings bullets below. Settle this before mapping a single money field – issuing twice (once natively, once from the ERP) is the costliest mistake in this section, and both modes are real deployments.

NATIVE MODE

Zid generates the tax invoice for the Order (`invoice_number` / `invoice_link` on the Order), computes the QR data shown on Credit Notes (`zatca.qr_code_data`, nullable when VAT is off, QR is disabled, or no tax number is set), and posts the document toward ZATCA itself, tracking `Posting to zatca` → `Posted to zatca` / `Zatca sending failed` as Order Tags you can read.

ERP duties: ingest `invoice_number` / `invoice_link`, QR payloads, invoice type/translated type, issue reason, and `original_invoice_number` on notes; reconcile your sales journal against Zid's numbers without creating local tax documents.

ERP-SIDE MODE

The ERP (or the merchant's own ZATCA connector) generates and clears/reports the e-invoice from Order data, then reports back: document id, number, QR, and sync sequence land on the Order (`merchant_invoicing_invoice_id/number/qr/sync_sequence`) and on Credit Notes (`zatca_credit_note_meta`: invoice id, credit-note number, sync sequence, invoice QR), and the status webhooks flip the same tags.

ERP duties: build the e-invoice from the Order payload (lines with net/gross/additions/tax exactly as invoiced – see **\$1.6**), clear/report it through your ZATCA path, then write back the document id, number, QR, and sequence so both systems agree; carry `original_invoice_number` onto every credit note you raise.

Credit Notes mirror the split exactly, with `original_invoice_number` linking each note to its source document. In both modes, B2B sales need the buyer block (`type: business` + name + tax number + commercial registration) – that is what makes the buyer's input-VAT claim possible.

How to detect the mode per store. Read `invoice_settings.is_zid_invoice_generation_enabled` on a full Order: true means native generation is on – the ERP must record, reference, and reconcile, never re-issue. When external documents exist instead, you will find the `merchant_invoicing_*` fields populated and ZATCA status tags on the Order. E-invoicing eligibility itself is per sales channel and device (storefront/API/mobile channels share one unit family; POS is per inventory location), so one merchant can legitimately run native on one channel and external on another – decide per channel, not per merchant.

Sequence and retry discipline. Every posted document consumes a ZATCA sequence number (`sync_sequence`), including retries of failed creates – a blind retry loop burns sequences and can force-delete a draft remotely with nothing resumable left. Treat e-invoice submission like a payment: idempotent-submit where the protocol allows, reconcile-then-retry everywhere else, and alert on `Zatca sending failed` the same day.

NOTE

Still confirm per deployment (ask the merchant or Zid before go-live): how the ERP registers as the external invoicing party for a store (onboarding path), which ZATCA phase each document type follows (clearance vs reporting/simplified), and who owns EGS-unit provisioning per channel. Do not assume these from the Order payload.

VAT settings, payments, and finance fields

- **The VAT settings endpoint exposes the whole mapping:** gross-vs-net catalog flag, shipping taxability, per-country rates/numbers/certificates with fallback, package gating. No change events – re-read before tax-sensitive processing.
- **Payment methods carry fees** (a COD fee is a real cost line) and visibility depends on delivery/provider configuration – re-read, don't cache forever. See also [Get Payment Options](#). Order-to-payment is not one-to-one; `can_request_order_due_payment` signals an outstanding balance.
- **Capture for finance per Order:** `id`, `code`, `invoice_number`, `issue_date` / `created_at` / `delivered_at` (different periods), `source_code`, marketplace/reseller flags, `edits_count` (a reason totals changed), fraud flag, weight details, `return_policy`. Never log the digital-products access token.
- **Nightly per Order:** line arithmetic, line-sum-to-header, invoice- `total` match, after-discount line-level VAT match, currency-intent check, Credit Note totals with linkage, wallet movements against `transaction_id`, gateway settlement net of fees to a tolerance. Trust the source total; add a rounding line instead of correcting it.

PART 3 • BUILDING IT

3.8 Testing, cutover, go-live

KEY TAKEAWAY

Never run a first pass against production: create allocates stock, Reverse confirms a Credit Note, refunds can invoke providers, and wallet credits move real money. Use a development store.

Set up a development store as described in [Create your First App](#).

Test matrix

Auth & headers

Full OAuth flow on a fresh store including `state` validation and a replayed callback; both tokens under distinct names; deliberate header swaps across families; missing `Role: Manager`; refresh; uninstall. A scope change on an installed store, proving you surface a reconnect prompt instead of retrying the `403`.

Catalog & customers

Every product class in scope; variants; options + input fields with prices; multi-location and infinite stock; real Arabic content; documented limits (32-char names, phone conflicts expecting `409`, tag writes from two writers).

Orders

Storefront Orders end to end; list with and without full payload (code must fail loudly on missing lines); allowed and disallowed status transitions; custom sub-status; non-shipping rejecting `indeliverly`; validate-then-create; multi-currency; B2B.

Returns & wallets

Full return, second partial, mixed classification buckets with separate stock/ `refund_total` verification, provider-handled vs merchant-managed refunds posting differently, duplicate-refund rejection. Wallet credit, debit, expiring credit, bulk partial success, and a non-GCC or feature-disabled store expecting `403`.

Discounts & VAT

Every in-scope discount mechanism with multi-discount both on and off (prove your reconciliation survives largest-wins suppression); a card/online-payment promotion asserted as not a bundle; VAT-inclusive and exclusive; additions included in line values; grouped products counted once.

Failure injection

Expired tokens, `429`, receiver `500` / `206` / timeout, duplicate and out-of-order delivery, bad signature, write timeouts proving search-before-resend and no double refund or double credit, offline downstream with paced drain.

Cutover

Freeze master data both sides; final load + reconcile; register webhooks and verify Healthy; enable delta one direction, verify, then the other; process a few real Orders under observation; release. **To stop safely**, keep acknowledging and queueing while pausing downstream – disabling the receiver degrades subscriptions toward Broken, and missed events never replay.

Monitor

Token-expiry countdown and refresh success per installation; webhook health; dead-letter depth; `401` / `429` rates; sync lag per object; reconciliation variance.

Escalating to Zid

Send store ID + App ID, endpoint or exact event name, redacted cURL, full response with status/body/ `X-Request-Id` , timestamp with timezone, expected behavior with a link to the reference page, and frequency. **Redact all credentials completely; never paste customer payloads.** Channels: the [Partner Help Center](#), the [Zid developer Slack workspace](#), and your [Partner Dashboard](#). Platform changes are announced in the [changelog](#) – subscribe to it.

Go-live checklist

- | | |
|--|--|
| ☐ Eligibility confirmed | ☐ Signature verified over raw bytes; <code>200/201/202/204</code> on all ack paths |
| ☐ Install journey matches the Activation & OAuth Policy (<code>state</code> validated, OAuth before onboarding, no token pasting, Service-ready gated on real readiness) | ☐ Idempotency reserved before business work; search-before-resend on every non-idempotent write |
| ☐ Scopes final, minimal and consented | ☐ Full payload wherever lines or money are needed |
| ☐ Tokens stored separately with mapping documented | ☐ Cross-reference backed up; initial load reconciled |
| ☐ Per-family request wrapper (incl. <code>Role: Manager</code> and <code>Store-Id</code>) | ☐ Reconciliation scheduled and reporting; alerting live |
| ☐ Refresh job scheduled off <code>expires_in</code> and run once; named owner recorded | ☐ Rollback documented and tested |

A

APPENDIX

WORKED EXAMPLES AND CROSS-REFERENCE

A.1 Worked examples

A.2 Cross-reference: concern → section

APPENDIX

A.1 Worked examples

KEY TAKEAWAY

Test values from a development store. Your store's IDs, codes, and totals will differ; the shapes will not.

Plain + variant order

Lines VAT-inclusive 138.00 (net 120.0025) + 64.98 (net 56.50): subtotal 176.50, VAT 26.48, shipping 15.00, COD 5.00, total 222.98. Invoice codes: sub_totals_before_vat, taxable_amount, vat, sub_totals_after_vat, shipping, zid_cod, total.

Single coupon order

ERP10PCT, 10% on 120.0025: discount -12.00025, VAT 16.2003375 (after discount), total 144.2025875. Line shows discount_percentage: 0.09999... with discounts.coupon.percentage: 10.

Triple coupon order

freeship = 10% + free shipping + free COD on a 79.99 sale line: coupon -7.999, free_shipping_coupon -15.00, coupon_cod_discount -5.00, VAT 10.79865 (after discount), total 82.78965. The coupon's own discount (-27.9990) is the sum of all three effects.

Bundle order

15%-off-one on a 79.00 net line: products_discount -11.85, VAT 29.721 (after discount), total 247.861. Line carries discount_rule with percentage: 15 and is_discounted: true.

Automatic progressive coupon

Subtotal $\geq 200 \rightarrow 10\%$, capped: self-applies with an on-cart progress meter, appears in coupon / coupons[] with no code typed. Digital lines (downloadable + voucher) are is_taxable: false: taxable_amount: 0 against subtotal 255.591, vat: 0, no shipping line, requires_shipping: false.

Reverse lifecycle

Delivered single-line order: calculate quotes the 5-line model $\rightarrow$ create (reason object) $\rightarrow$ classify received $\rightarrow$ refund_total: 138 $\rightarrow$ bank-transfer refund recorded as amount: "138.00" $\rightarrow$ second refund attempt rejected $\rightarrow$ order reversed. No credit note surfaced on unpaid orders — notes follow paid money.

APPENDIX

A.2 Cross-reference: concern → section

ERP concern	Section	Page
Which data pushes vs polls, exact event names	§1.1	7
Base URLs, which headers per API family, 401s	§1.2	9
OAuth, state, token storage and expiry	§1.3	10
Install flow, activation policy, app review	§1.4	12
Scopes, 403s, eligibility	§1.5	13
Order/invoice totals, VAT math, rounding	§1.6 , §3.7	15, 45
VAT rate per country, non-Saudi stores	§1.6	15
Multi-currency, order vs store currency	§1.6a	17
When the invoice is generated, invoice settings	§3.7	45
Status mapping, returns trigger	§1.7 , §3.3	18, 40
Product/variant/stock sync, catalog load	§1.8 , §3.1	19, 32
Customer matching, B2B fields, tags	§1.9	20
Any discount question	§1.10	21
Store credit, wallet balances, cashback	§1.11	24
Marketplace order typing, invoicing, returns, PII	§1.12	26
Sync direction, ownership, reconciliation design	§2.1	29
Mapping Zid objects to ERP objects	§2.2	30
Order pull, order create, fulfillment locations, tags/comments	§3.2	33
Which fields are on List vs View vs webhook, and what to map	§3.2a	35
JSON types, string-vs-number money, UTC timestamps	§1.6	15
Coupons, bundles, progressive rules	§3.4	41
Webhook receiver, signature code, retries, idempotency	§3.5	42
Errors, limits, silent gaps	§3.6	44
Invoice issuer, VAT config, finance fields	§3.7	45
Testing, cutover, go-live, escalation	§3.8	48



Build with Zid

API REFERENCE

docs.zid.sa

PLATFORM CHANGELOG

changelog.partner.zid.sa

PARTNER HELP CENTER

help-partner.zid.sa

PARTNER DASHBOARD

partner.zid.sa

Version 1.2.0 – last reviewed 7 October 2026 against the Zid platform, a live development store, and an observed webhook delivery. Platform behavior moves: re-verify figures against the API reference and the changelog before relying on them, and record the date you verified.

ZID-ZAM-DG-ERP • Partner distribution • Not tax, legal, or compliance advice.