Local Development & Accounting Events Setup

Quick Start (SQLite Fallback)

```
cd backend
uvicorn main:app --reload
```

If Postgres is unreachable / not configured the app falls back to `./dev.db` (SQLite + aiosqlite).

Verify live DB connectivity:

```
curl http://127.0.0.1:8000/api/v1/healthz
```

Response fields:

- `ok`: true if a `SELECT 1` succeeded
- `fallback`: true if using SQLite instead of primary DATABASE_URL

Run smoke tests:

```
bash scripts/smoke_tests.sh
```

Using Postgres (Docker)

Create a `.env` file (copy `.env.example`) and set:

```
DATABASE_URL=postgresql+asyncpg://postgres:postgres@localhost:5432/odoo_
hack
```

Start a local Postgres (see `docker-compose.db.yml` once added):

```
docker compose -f docker-compose.db.yml up -d
```

Then start API:

```
uvicorn main:app --reload
```

Installing Dependencies

Project uses `pyproject.toml` (PEP 621). Example with uv:

```
pip install uv
uv pip install -e .[dev]
```

or plain pip:

```
pip install -r <(python -c 'import
tomllib,sys;d=tomllib.load(open("pyproject.toml","rb"));print("\n".join(
d["project"]["dependencies"]))')
```

Transaction Event Ingestion (Replaces Kafka Prototype)

We simulate a Kafka consumer via an HTTP endpoint:

POST `/events/transaction`

Single event example (`event.json`):

```
{
    "txn_id": "evt-20250920-0001",
    "ref_type": "INVOICE",
    "ref_id": "inv-0001",
    "date": "2025-09-20T12:34:56Z",
    "description": "Sale of 5 office chairs",
    "entries": [
        {"account_code": "1100", "debit": "5000.00", "credit": "0.00"},
        {"account_code": "4000", "debit": "0.00", "credit": "5000.00"}
    ],
    "meta": {"source": "api-testing"}
}
```

Send it:

```
TOKEN=... ./scripts/post_event.sh event.json
```

Batch mode: supply an array `[ {...}, {...} ]` in the JSON file.

Validation rules:

- Entries must balance (sum debit == sum credit)
- Each line: exactly one of debit/credit > 0

- Account codes must exist (400 if not found)
- Idempotent on `txn_id` (duplicate returns status `already_processed`)

Accounting Reports

Endpoints (JWT protected):

- `GET /reports/pnl?start=YYYY-MM-DD&end=YYYY-MM-DD`
- `GET /reports/balance_sheet?as_of=YYYY-MM-DD`

Sign convention documented in code (`crud/reports.py`).

Running Tests

```
pytest -q
```

Planned integration test (to add):
`shiv_accounts_cloud/tests/test_transaction_reporting.py` seeds accounts, posts purchase & sales flows (Inventory + COGS + Sales) and asserts P&L & Balance Sheet integrity.

Database Behavior

- First tries primary `DATABASE_URL` (Postgres expected).
- On failure, logs silently (current minimal impl) and switches to SQLite file `dev.db`.
- When in fallback, schema auto-created via existing startup hook.
- Now emits a warning log `Primary database unavailable, switching to SQLite fallback: <error>`.

Health Endpoints

- `/health`: Static application status.
- `/api/v1/healthz`: Live DB ping + fallback flag.

Common Issues

- Missing async driver: ensure `asyncpg` and `aiosqlite` installed (present in `pyproject.toml`).
- Stale `dev.db`: Delete file to recreate schema.

Next Enhancements (Optional)

- Background queue for event ingestion
- Aggregated cached materialized views
- Segment / dimension reporting (by product, customer)
- Duplicate txn replay audit log

Purchase & Sales Transactions API (In-Process Kafka Replacement)

Endpoints:

- `POST /transactions/purchase_order` (JWT required)
- `POST /transactions/sales_order` (JWT required)
- `GET /transactions/{txn_id}`

Request Examples:

```
// Purchase Order
{
    "vendor_id": "VENDOR1",
    "items": [
        {"product_id": "p1", "qty": "2", "unit_price": "100.00",
"tax_percent": "5"},
        {"product_id": "p2", "qty": "1", "unit_price": "50.00",
"tax_percent": "5"}
    ],
    "order_date": "2025-09-20T00:00:00Z",
    "expected_receipt_date": "2025-09-25T00:00:00Z",
    "txn_id": "PO-SAMPLE-1"
}
```

```
// Sales Order
{
    "customer_id": "CUSTOMER1",
    "items": [
        {"product_id": "p1", "qty": "3", "unit_price": "19.99",
"tax_percent": "10", "discount": "1.00"}
    ],
    "order_date": "2025-09-20T00:00:00Z",
    "txn_id": "SO-SAMPLE-1"
}
```

Response (201 or 200 if idempotent re-post):

```
{
    "transaction": {"id": "...", "txn_id": "PO-SAMPLE-1", "type":
"PURCHASE_ORDER", "status": "POSTED", "total_amount": 315.00, ...},
    "journal_summary": {"total_debit": 315.00, "total_credit": 315.00,
"balanced": true},
    "dispatch_status": {"status": 200, "attempt": 1}
}
```

Double-entry lines are generated internally and dispatched as an event (simulating Kafka) to `/events/transaction`.

Draft Posting Flow

If you set `auto_post` (PO) or `invoice_on_confirm` (SO) to `false`, the transaction is stored in `DRAFT` with original items snapshot in `transaction.meta._draft_items`. Later you can finalize via:

```
POST /transactions/{txn_id}/post
```

This regenerates journal entries and posts the transaction.

Payment Transactions

Endpoint:

```
POST /transactions/payment
```

Request example:

```
{
    "direction": "in",  // in = receipt, out = payment
    "amount": "150.00",
    "payment_date": "2025-09-20T00:00:00Z",
    "party_type": "customer",
    "party_id": "CUST-123",
    "meta": {"note": "Settlement"}
}
```

Response mirrors order endpoints. Payments are always created in `POSTED` status. Idempotent if you include `txn_id` field.

Event Enrichment & Logging

Each dispatched event now includes `tenant_id` and `user_id` for downstream multi-tenant correlation. Dispatcher outputs structured JSON logs to stdout with keys: `component=event_dispatcher`, `state` (success|error), `attempt`, `txn_id`, `ref_type`, `ref_id`, `tenant_id`, plus `status_code` or `error`.

Immediate Journal Persistence (Optional)

Set `PERSIST_JOURNAL_IMMEDIATE=true` to persist `JournalEntry` + `JournalLine` records synchronously at transaction creation (in addition to dispatch). Idempotent on `txn_id` to avoid duplicates.

Account Mapping Overrides

Default account codes reside in code. To override, create `backend/config/account_mappings.json` (see `account_mappings.example.json`). Example:

```
{
    "cash_account": "1010",
    "sales_account": "4100"
}
```

On next process start the overrides merge with defaults. (Current implementation requires restart; future hot-reload could clear internal cache.)

Troubleshooting

Scenario	Symptom	Resolution
Idempotent collision	201 response but `dispatch_status: idempotent` unexpectedly	Ensure client reuses `txn_id` only for true retries; generate new UUID otherwise
Dispatch errors	`dispatch_status.status` = `error` with `error` message	Check connectivity to ingestion URL, review dispatcher stdout logs (search `component=event_dispatcher state=error`)
Unbalanced event rejection	400 from `/events/transaction`	Verify debit == credit; run `tests/test_accounting_engine.py` to confirm generator logic
Missing account code	400 detail `Account code XYZ not found`	Add chart of accounts seed (or override mapping) so code exists before posting
Draft posting fails	400 `Draft items snapshot missing`	Original draft meta removed; recreate draft transaction

Performance Benchmark (Planned)

Placeholder script will compare sequential vs potential future batch ingestion timings. Not yet implemented.

Environment Variables

Variable	Default	Meaning
`USE_INTERNAL_POST`	`true`	If true, POST events to local ingestion endpoint
`PROCESS_LOCALLY`	`false`	If true and direct consumer callable available, bypass HTTP
`CONSUMER_HOST`	`localhost`	Host for event ingestion

Variable	Default	Meaning
CONSUMER_PORT	8000	Port for event ingestion

Scripts

Use helper scripts (require TOKEN):

```
export TOKEN="<jwt>"
bash scripts/post_purchase.sh
bash scripts/post_sale.sh
```

Idempotency

Supply txn_id to ensure re-submission returns existing transaction (status 200/201 with dispatch_status: idempotent).

Testing

```
pytest tests/test_accounting_engine.py -q
pytest tests/test_transactions_flow.py -q
```

If dispatch endpoint not reachable, response will include an error status; journal still persisted (Transaction row). Handle retries externally if needed.