relation

```
[VENDOR] → [PO] → [GRN] → [STOCK IN] → [LEDGER]
                                ↓
                          [PURCHASE RETURN] → [STOCK OUT] → [LEDGER
```

🔗 Full Relationship Explanation

1. VENDOR → PURCHASE ORDER (PO)

- **Entity**: `Vendor`, `PurchaseOrder`
- **Relation**: A PO is created for a specific Vendor.
- **Fields**:
  - `purchaseOrder.vendor`: references a `Vendor._id`
- **UI**: In the "Create PO" screen, vendor is selected when generating a PO.
- **Usage**: Defines what items and quantities you intend to buy from a vendor.

---

2. PO → GOODS RECEIPT NOTE (GRN)

- **Entity**: `GoodsReceipt`
- **Relation**: GRN is tied to a specific PO.
- **Fields**:
  - `goodsReceipt.purchaseOrder`: references `PurchaseOrder._id`
- **UI**: In the "GRN / Goods Receipt" screen, you select a PO.
- **Usage**: When goods are received against a PO, a GRN records what was actually received (could be partial or full).

---

3. GRN → STOCK IN

- **Entity**: `Stock`
- **Logic**: For each item in GRN:
  - `increaseStock(item, receivedQty)` is called.
- **Fields**:
  - `Stock.item`: reference to `Item._id`
  - `Stock.quantity`: incremented by `receivedQty`
- **Result**: The stock level is updated to reflect received goods.

---

4. STOCK IN → LEDGER

- **Entity**: `StockLedger`
- **Trigger**: In `increaseStock(...)`, a ledger entry is created.
- **Fields**:
  - `transactionType`: `"IN"`
  - `source`: `"GoodsReceipt"`
  - `sourceId`: the `GRN._id`
- **Usage**: This creates an auditable trail of every stock inflow.

🔁 PURCHASE RETURN FLOW

5. PO → PURCHASE RETURN

- **Entity**: `PurchaseReturn`
- **Relation**: Purchase Return is linked to a previous PO.
- **Fields**:
  - `referenceId`: references `PurchaseOrder._id`
- **Usage**: If items are defective or over-delivered, they can be returned to the vendor.

---

6. PURCHASE RETURN → STOCK OUT

- **Logic**: On purchase return:
  - `decreaseStock(item, qty)` is called.
- **Result**: Stock is reduced to reflect returned quantity.

---

7. STOCK OUT → LEDGER

- **Entity**: `StockLedger`
- **Trigger**: In `decreaseStock(...)`, a ledger entry is created.
- **Fields**:
  - `transactionType`: `"OUT"`
  - `source`: `"PurchaseReturn"`
  - `sourceId`: the `PurchaseReturn._id`
- **Usage**: Keeps a log of why stock was decreased and links it to the return.

✅ Summary of Flow Implementation Status

Step	Entity	Implemented?	Remarks
1 → 2	`PurchaseOrder → GoodsReceipt`	✅	Done
2 → 3	`GoodsReceipt → Stock`	✅	via `increaseStock`

3 → 4	`Stock → StockLedger`	✅	via `StockLedger.create`
2 ↓ 5	`PO → PurchaseReturn`	✅	via `PurchaseReturn.referenceId`
5 → 6	`PurchaseReturn → Stock`	✅	via `decreaseStock`
6 → 7	`Stock → StockLedger`	✅	via `StockLedger.create` with `"OUT"`

🔧 Suggested Enhancements

1. **PO Remaining Logic for Return**: Optionally, you could check if returned items do not exceed received quantities.
2. **Return Reason Enum**: Consider adding enums for common reasons (`damaged`, `expired`, `excess`) in PurchaseReturn.
3. **Ledger View Enhancements**:
   - Add `source` and `sourceId` filters.
   - Group by item or source for reporting.

🔗 Sales Flow: End-to-End Explanation

1. CLIENT → SALES INVOICE

- **Entity**: `Client`, `SalesInvoice`
- **Relation**: A sales invoice is created for a specific client.
- **Fields**:
  - `SalesInvoice.client` → references `Client._id`
- **Usage**: This records the sale of items to a client.
- **UI**: In the "Create Sales Invoice" form, you select the client and items.

---

2. SALES INVOICE → STOCK OUT

- **Entity**: `Stock`
- **Logic**: For each item in the invoice:
  - `decreaseStock(item, quantity)` is executed.
- **Result**:
  - `Stock.quantity` is reduced.
  - The inventory reflects that these goods have been sold.

---

3. STOCK OUT → LEDGER

- **Entity**: `StockLedger`

- **Trigger**: Inside `decreaseStock(...)` function.
- **Fields**:
  - `transactionType`: `"OUT"`
  - `source`: `"SalesInvoice"`
  - `sourceId`: the corresponding `SalesInvoice._id`
- **Usage**: This forms the audit trail of every stock movement related to sales.

🔁 Sales Return Flow

4. SALES INVOICE → SALES RETURN

- **Entity**: `SalesReturn`
- **Relation**: A return is linked to an existing Sales Invoice.
- **Fields**:
  - `referenceId` → references `SalesInvoice._id`
- **Purpose**:
  - Client returns sold items due to reasons like damage or incorrect product.
- **UI**: In "Sales Return", you select invoice, see items, and specify return quantities.

---

5. SALES RETURN → STOCK IN

- **Logic**: For each item returned:
  - `increaseStock(item, quantity)` is executed.
- **Result**:
  - Stock is incremented.
  - Returned goods are added back into inventory.

---

6. STOCK IN → LEDGER

- **Entity**: `StockLedger`
- **Trigger**: Inside `increaseStock(...)` in `salesReturnController.js`
- **Fields**:
  - `transactionType`: `"RETURN"`
  - `source`: `"SalesReturn"`
  - `sourceId`: the corresponding `SalesReturn._id`
- **Purpose**: Maintains traceability of stock increases due to returns.

✅ Summary of Implementation Status

Step	Entity	Status	Remarks
1 → 2	`Client → SalesInvoice`	✅	In `createInvoice()`
2 → 3	`SalesInvoice → Stock (OUT)`	✅	Done via `decreaseStock`
3 → 4	`Stock → Ledger`	✅	`"OUT"` entry via `decreaseStock()`
1 ↓ 5	`SalesInvoice → SalesReturn`	✅	Via `referenceId` in return
5 → 6	`SalesReturn → Stock (IN)`	✅	via `increaseStock()`
6 → 7	`Stock → Ledger`	✅	`"RETURN"` entry logged

🔧 Optional Enhancements

A. Validation of Sales Returns

✔️ Already implemented:

- Prevents over-returning items (i.e., cannot return more than sold - already returned).

B. Sales Return Reasons

You may add:

```
js
CopyEdit
reason: {
  type: String,
  enum: ["damaged", "expired", "wrong item", "client request"],
}
```

C. Reporting View Suggestions

You can build a consolidated ledger/report view by:

- Filtering `StockLedger` by `source = "SalesInvoice"` or `"SalesReturn"`
- Grouping by item, client, or invoice

📦 Delivery Challan Stock Flow

1. DELIVERY CHALLAN → STOCK OUT

✅ Entity Involved

- **Model**: `DeliveryChallan.js`
- **Controller**: `deliveryChallanController.js`

✅ What Happens

When a delivery challan is created:

- The items and quantities listed in it are fetched.
- For each item, stock is **decreased** using:

```
js
CopyEdit
await decreaseStock(i.item, i.quantity);
```

✅ Function Used

- `decreaseStock(itemId, quantity, remarks?, updatedBy?)`
  - Finds the item's current stock
  - Decreases it by the specified quantity
  - Sets `lastUpdatedBy` and `remarks`

---

2. STOCK OUT → LEDGER

✅ Entity Involved

- **Model**: `StockLedger.js`
- **Triggered from**: `decreaseStock()` in `stockController.js` or `stockHelpers.js`

✅ What's Recorded

- A new `StockLedger` entry is created **automatically**
- Example:

```
js
CopyEdit
await StockLedger.create({
  item,
  transactionType: "OUT",
  quantity,
  source: "DeliveryChallan",
  sourceId: deliveryChallan._id,
  timestamp: new Date(),
});
```

✅ Fields

- `transactionType`: `"OUT"`
- `source`: `"DeliveryChallan"`
- `sourceId`: reference to `DeliveryChallan._id`

- `quantity`: item quantity
- `timestamp`: date of issue

🔄 Complete Flow Summary

Step	Entity	Status	Description
1	**Delivery Challan**	✅ Implemented	Created from selected client + item list
2	**Stock OUT**	✅ Automatic via `decreaseStock()`	Quantity reduced
3	**Ledger Entry**	✅ Logged via `StockLedger`	With `source = "DeliveryChallan"`

✅ Business Meaning

- The **Delivery Challan** acts as proof of goods dispatch **without billing** (yet).
- Used in B2B transactions where delivery precedes invoice.
- This ensures **stock is reduced** upon delivery, regardless of billing stage.
- The **ledger** entry guarantees auditability and traceability of outbound stock.

Let me know if you want:

- 🔍 A report or view of all stock movements triggered by delivery challans.
- 📤 Auto-convert a DC to Sales Invoice later.
- 🧾 Challan PDF generation for dispatch.

Here is the complete breakdown of the inventory flow:

> [STOCK ADJUSTMENTS] → IN / OUT → [LEDGER]

This mechanism handles **manual corrections** to inventory levels — for example, discrepancies found during physical stock audits.

⚙️ STOCK ADJUSTMENTS

✅ Entity Involved

- **Model**: `StockAdjustment.js`
- **Controller**: `stockAdjustmentController.js`
- **Route**: `POST /api/stock-adjustments`

✅ What Happens

When a stock adjustment is submitted:

- The system reads:
  - `item`
  - `adjustmentType` (`"increase"` or `"decrease"`)
  - `quantity`
  - `reason`
- Then:
  - Calls either:
    - `increaseStock(item, qty)` **or**
    - `decreaseStock(item, qty)`
  - Logs the adjustment in `StockAdjustment` collection

🔁 Example Code:

```
js
CopyEdit
if (adjustmentType === "increase") {
  await increaseStock(item, quantity);
} else {
  await decreaseStock(item, quantity);
}
```

📜 LEDGER ENTRY

✅ Automatically Logged in `StockLedger`

- After adjusting stock, the system creates a ledger entry:

```
js
CopyEdit
await StockLedger.create({
  item,
  transactionType: "ADJUST",
  quantity,
  source: "StockAdjustment",
  sourceId: adjustment._id,
  timestamp: new Date(),
});
```

✅ Fields Logged

Field	Value
`item`	Referenced item
`transactionType`	`"ADJUST"`
`quantity`	+ / –
`source`	`"StockAdjustment"`
`sourceId`	`_id` of the adjustment
`timestamp`	Current date

📋 Summary of Flow

Step	Action	Source	Result
1️⃣	Manual stock fix	`StockAdjustment`	Record of correction
2️⃣	Increase/Decrease stock	`increaseStock()` / `decreaseStock()`	Adjust central stock
3️⃣	Ledger log	`StockLedger.create()`	Movement is traced and auditable

📌 Use Cases

- **Found extra stock** in warehouse → Adjustment **IN**
- **Lost/damaged** items → Adjustment **OUT**
- **Cycle count mismatch** → Adjust to match reality

Let me know if you'd like:

- A **filter view**: "Only Adjustments in Ledger"
- A toggle between **Increase/Decrease** color-coded
- Validation to prevent unauthorized stock corrections