Title: Database Schema Audit & Improvement – ERP SystemReporting
To: Harsh Maniar
Submitted by: Vraj lalwala & Naman Shukla
Deadline: Monday(First Half)

Database Schema improvements suggestion

Problems	Why it's an issue	Suggested fix	Expected benefit
SalesOrder and PurchaseOrder have two separate collections	Sales and purchase orders share a lot of the same information (like items, total amount, status). Combining them means less duplicate code and easier management.	• Instead of having separate collections for SalesOrder and PurchaseOrder, we can have a single invoices collection. This new collection has a type field which will be either 'sales' or 'purchase' to tell them apart. • We can have a clever partner field that can link to either a customer (for sales) or a supplier (for purchases) dynamically. This avoids having two separate, often empty, fields for customer and supplier IDs.	• Less duplication • Simpler Reporting

No direct account creation option for user	Many times users don't want to directly sign in with auth providers.	Add your own Authentication option too and store hashed password, first name , last name, etc. in collections.	• Flexible login • Better UX
Inventory and Stock Ledger has two separate collections	Updating an Inventory record and adding a StockLedger entry can be tedious. We have to worry about the "current inventory" number getting out of sync with the actual transactions.	We can get rid of the separate Inventory and StockLedger . Now, there's just one collection called stock_movements. Every single time stock moves in or out, or is adjusted, a new record is added here. To find out your current stock, you'd simply add up all the relevant movements.	• Always Accurate Stock • Full Audit Trail • Simpler Updates
Not all collections have created_at and updated_at fields	Without timestamps, it's impossible to track when records were added or modified. This limits your ability to audit user actions, debug issues, measure performance over time, or clean up stale data. It also weakens accountability, especially in multi-user or multi-vendor ERP environments.	Added createdAt and UpdatedAt fields to almost all collections	• Track when and by whom a record was created or changed. • Easily filter records based on age for cleanup, archiving, or alerts. • Identify slow-changing vs. high-activity records for optimization. • Support time-based queries like "Sales in the last 7 days".

Missing connection between products and unit_masters	Since there is no direct link to `unit_masters`, there's no guarantee that the value stored in `products.unit` actually exists in `unit_masters`. Typos like `"PCS"` or `"pce"` would go undetected. If you want to rename a unit or change its label (e.g., `"pcs"` to `"pieces"`), you would have to update all `products` manually. This violates normalization principles. You cannot enforce that only valid, predefined units are used in products unless you implement custom logic. Without a central definition, units could appear in different formats across the app (`"KG"` vs `"kg"`), confusing users.	Give one to many relationship between unit_masters.unitId and products.unit	• Only valid units from unit_masters can be used, reducing errors and inconsistencies. • Units are managed in one place. If you rename `"pcs"` to "pieces"in unit_masters, it automatically reflects in all associated products. • You can show display names, abbreviations, or even multilingual labels for each unit from a centralized collection. • APIs can return unit metadata (`unitName`, `unitSymbol`) by simply populating the `unitId`, improving clarity for front-end developers.

No reverse references (e.g., products to suppliers, customers to invoices)	Hard to query related data efficiently	Add arrays or virtual refs, or use Mongoose population	• Easier data retrieval, improved query performance
No Collection for shops meaning multiple shops feature can't be implemented	An owner can't have all it's shops listed under his single vendor id	Adding a new collection that stores shop data which can be referenced by vendor	• Easier to manage • More control • Clear separated data
No Settings schema for stores	No dynamic tax handling, no timezone awareness, no custom invoice format, hardcoded settings , etc.	Add a setting schema which references store.	• Dynamic invoice formatting • Tax configuration • Flexibility
No notification service for store	Stores can't tell their customers about any event or status of their product delivery or any issues.	Add a notification schema which references store and its customers.	• Better communication • Better engagement • Better UX
No restock suggestion to store vendors.	On the stock getting finished there is no communication to store.	Add a restock suggestion schema which help us achieve this.	• Better maintainability • Better trust
No report schema .	Not having it leads to manual labor, fragility, and operational inefficiency, especially in multi-store or multi-user systems	Add a report schema with all necessary fields.	• Centralized report system • Automated scheduling • Custom report support

Updated Schema Mock

```
const mongoose = require('mongoose')
const { Schema } = mongoose

const UserSchema = new Schema({
  email: { type: String, required: true, match: /^[\w._%+-]+@[\w.-]+\.[a-zA-Z]{2,}$/, unique: true },
```

```
  password: { type: String, required: true, minlength: 8 },
  firstName: { type: String, required: true },
  lastName: { type: String, required: true },
  phone: String,
  role: { type: String, required: true, enum: ["super_admin", "store_manager", "sales_staff",
"inventory_manager", "accountant"] },
  permissions: {
    canAccessAllStores: Boolean,
    canManageUsers: Boolean,
    canManageProducts: Boolean,
    canCreateInvoices: Boolean,
    canManageStock: Boolean,
    canViewReports: Boolean,
    canManageSettings: Boolean
  },
  assignedStores: [{ type: Schema.Types.ObjectId, ref: 'Store' }],
  status: { type: String, enum: ["active", "inactive", "suspended"], required: true },
  lastLogin: Date,
  createdAt: { type: Date, default: Date.now },
  updatedAt: { type: Date, default: Date.now },
  createdBy: { type: Schema.Types.ObjectId, ref: 'User' }
});

UserSchema.index({ email: 1 }, { unique: true });
UserSchema.index({ role: 1, status: 1 });
UserSchema.index({ assignedStores: 1 });

module.exports = mongoose.model('User', UserSchema);


const ReportSchema = new Schema({
  name: { type: String, required: true },
  type: {
    type: String,
    enum: ['sales', 'inventory', 'financial', 'customer', 'custom'],
    required: true
  },
  parameters: Schema.Types.Mixed,
  schedule: Schema.Types.Mixed,
['email@example.com'], isActive: true }
  lastGenerated: Date,
  createdAt: { type: Date, default: Date.now },
  createdBy: { type: Schema.Types.ObjectId, ref: 'User' }
});
```

```
ReportSchema.index({ type: 1 });
ReportSchema.index({ createdBy: 1 });
ReportSchema.index({ createdAt: -1 });

module.exports = mongoose.model('Report', ReportSchema);


const RestockSuggestionSchema = new Schema({
  productId: { type: Schema.Types.ObjectId, ref: 'Product', required: true },
  storeId: { type: Schema.Types.ObjectId, ref: 'Store', required: true },
  suggestedQty: { type: Number, min: 0, required: true },
  currentStock: { type: Number, min: 0 },
  reorderPoint: { type: Number, min: 0 },
  createdAt: { type: Date, default: Date.now }
});

RestockSuggestionSchema.index({ productId: 1, storeId: 1 });

module.exports = mongoose.model('RestockSuggestion', RestockSuggestionSchema);


const ShopSchema = new mongoose.Schema({
  name: { type: String, required: true },
  address: { type: String },
  ownerId: { type: mongoose.Schema.Types.ObjectId, ref:'Vendor', required: true },
  contact: { type: String },
  settings: { type: mongoose.Schema.Types.Mixed }
}, { timestamps: true });

const UnitMasterSchema = new Schema({
  unitName: { type: String, required: true },
  unitSymbol: { type: String, required: true }
}, { timestamps: true });

const ProductSchema = new Schema({
  name: { type: String, minlength: 1 },
  description: String,
  sku: { type: String, unique: true, minlength: 1 },
  barcode: String,
  categoryId: { type: Types.ObjectId, ref: 'Category' },
  brand: String,
  model: String,
  price: { type: Number, min: 0 },
```

```
  cost: { type: Number, min: 0 },
 images: [{ url: String, alt: String, isPrimary: Boolean }],
  specifications: Schema.Types.Mixed,
  supplier: {
    name: String,
    contact: String,
    leadTime: Number,
    minimumOrder: Number
  },
  status: { type: String, enum: ['active', 'inactive', 'discontinued'] },
  createdAt: { type: Date, default: Date.now },
  updatedAt: { type: Date, default: Date.now },
  createdBy: { type: Types.ObjectId, ref: 'User' }
});

ProductSchema.index({ sku: 1 }, { unique: true });
ProductSchema.index({ categoryId: 1 });
ProductSchema.index({ status: 1 });
  currency: { type: String, default: 'USD' },

  suppliers: [{ type: Schema.Types.ObjectId, ref: 'Supplier' }]
}, { timestamps: true });

const SupplierSchema = new Schema({
  name: { type: String, required: true },
  contactInfo: {
    email: String,
    phone: String,
    address: String
  },
  products: [{ type: Schema.Types.ObjectId, ref: 'Product' }]
}, { timestamps: true });

const CustomerSchema = new Schema({
  customerCode: { type: String, unique: true },
  type: { type: String, enum: ['individual', 'business'] },
  firstName: String,
  lastName: String,
  companyName: String,
  taxId: String,
  email: String,
  phone: String,
  address: {
    street: String,
```

```
    city: String,
    state: String,
    zipCode: String,
    country: String
  },
  billingAddress: {
    street: String,
    city: String,
    state: String,
    zipCode: String,
    country: String
  },
  category: { type: String, enum: ['regular', 'vip', 'wholesale', 'retail'] },
  creditLimit: { type: Number, min: 0, default: 0 },
  paymentTerms: { type: Number, default: 30 },
  discount: { type: Number, min: 0, max: 1, default: 0 },
  status: { type: String, enum: ['active', 'inactive', 'blocked'] },
  notes: String,
  createdAt: { type: Date, default: Date.now },
  updatedAt: Date,
  createdBy: { type: Types.ObjectId, ref: 'User' }
});

CustomerSchema.index({ customerCode: 1 }, { unique: true });
CustomerSchema.index({ type: 1 });
CustomerSchema.index({ status: 1 });

const NotificationSchema = new Schema({
  userId: { type: Schema.Types.ObjectId, ref: 'User', required: true },
  storeId: { type: Schema.Types.ObjectId, ref: 'Store' },
  type: {
    type: String,
    enum: [
      'info', 'warning', 'error', 'success',
      'low_stock', 'overdue_invoice', 'system'
    ],
    required: true
  },
  title: { type: String, required: true },
  message: { type: String, required: true },
  data: Schema.Types.Mixed,
  isRead: { type: Boolean, default: false },
  readAt: Date,
  priority: {
```

```
    type: String,
    enum: ['low', 'medium', 'high', 'urgent'],
    default: 'medium'
  },
  expiresAt: Date,
  createdAt: { type: Date, default: Date.now }
});

NotificationSchema.index({ userId: 1, isRead: 1, createdAt: -1 });
NotificationSchema.index({ type: 1 });
NotificationSchema.index({ expiresAt: 1 });


const SettingsSchema = new Schema({
  storeId: { type: Schema.Types.ObjectId, ref: 'Store' },
  currency: { type: String, default: 'USD' },
  gstRate: { type: Number, default: 0 },
  invoicePrefix: { type: String, default: 'INV-' },
  contactEmail: { type: String },
  invoiceFooter: { type: String },
  language: { type: String, default: 'en' },
  timezone: { type: String },
  businessHours: Schema.Types.Mixed,
  createdAt: { type: Date, default: Date.now },
  updatedAt: { type: Date, default: Date.now }
});

SettingsSchema.index({ storeId: 1 });


const InvoiceSchema = new Schema({
  invoiceNumber: { type: String, unique: true },
  customerId: { type: Types.ObjectId, ref: 'Customer' },
  storeId: { type: Types.ObjectId, ref: 'Store' },
  items: [
    {
      productId: { type: Types.ObjectId, ref: 'Product' },
      productName: String,
      sku: String,
      quantity: { type: Number, min: 0.01 },
      unitPrice: { type: Number, min: 0 },
      discount: { type: Number, min: 0, default: 0 },
      total: { type: Number, min: 0 }
    }
```

```
  ],
  subtotal: { type: Number, min: 0 },
  discountAmount: { type: Number, min: 0, default: 0 },
  taxAmount: { type: Number, min: 0, default: 0 },
  taxRate: { type: Number, min: 0, max: 1 },
  total: { type: Number, min: 0 },
  currency: { type: String, default: 'USD' },
  paymentTerms: { type: Number, default: 30 },
  dueDate: Date,
  status: { type: String, enum: ['draft', 'sent', 'paid', 'partial_paid', 'overdue', 'cancelled'] },
  paymentStatus: { type: String, enum: ['unpaid', 'partial', 'paid', 'overpaid'] },
  notes: String,
  createdAt: { type: Date, default: Date.now },
  updatedAt: Date,
  createdBy: { type: Types.ObjectId, ref: 'User' }
});

InvoiceSchema.index({ invoiceNumber: 1 }, { unique: true });
InvoiceSchema.index({ customerId: 1 });
InvoiceSchema.index({ storeId: 1 });
InvoiceSchema.index({ status: 1 });
InvoiceSchema.index({ paymentStatus: 1 });


const StockMovementSchema = new Schema({
  productId: { type: Types.ObjectId, ref: 'Product' },
  fromStore: { type: Types.ObjectId, ref: 'Store' },
  toStore: { type: Types.ObjectId, ref: 'Store' },
  type: { type: String, enum: ['transfer', 'adjustment', 'restock', 'sale', 'return'] },
  quantity: { type: Number, min: 0 },
  reason: String,
  date: { type: Date, default: Date.now },
  performedBy: { type: Types.ObjectId, ref: 'User' }
});

StockMovementSchema.index({ productId: 1 });
StockMovementSchema.index({ fromStore: 1 });
StockMovementSchema.index({ toStore: 1 });
StockMovementSchema.index({ type: 1 });


mongoose.model('User', UserSchema);
mongoose.model('UnitMaster', UnitMasterSchema);
mongoose.model('Product', ProductSchema);
```

```
mongoose.model('Supplier', SupplierSchema);
mongoose.model('Customer', CustomerSchema);
mongoose.model('Invoice', InvoiceSchema);
mongoose.model('StockMovement', StockMovementSchema);
mongoose.model(‘ShopSchema’, ShopSchema);
```

Intern Reflection Note

During this schema audit, I learned how critical clean data modeling is for the scalability and maintainability of a system. Initially, the collections seemed functional, but on closer inspection, I noticed several opportunities to normalize fields, structure data, and define relationships more clearly. It was especially insightful to see how missing references and ambiguous field structures can affect real-world features like order tracking, tax calculation, and stock control. I also deepened my understanding of MongoDB design patterns.

Thanks